AD-A115 558 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
DESIGN AND IMPLEMENTATION OF A BACKEND MULTIPLE-PROCESSOR RELAT--ETC(U)
DEC 81 R W FONDEN
UNCLASSIFIED AFIT/GCS/EE/81D-6 NL

1 of 3
AD
A115558

DDC

①

AD A115558

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

82 06 14 160

AFIT/GCS/EE/81D-6

DESIGN AND IMPLEMENTATION OF A BACKEND MULTIPLE-PROCESSOR RELATIONAL DATA BASE COMPUTER SYSTEM

THESIS

AFIT/GCS/EE/81D-6 Robert W. Fonden
Captain USAF

AFIT/GCS/EE/81D-6

DESIGN AND IMPLEMENTATION OF A BACKEND MULTIPLE-PROCESSOR RELATIONAL DATA BASE COMPUTER SYSTEM

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

Robert W. Fonden, B.A.

Captain USAF

Graduate Computer Science

December 1981

Preface

This work presents a feasibility study, a requirements analysis, an initial design, and a first stage implementation of a backend multiple-processor relational data base computer system for the Digital Engineering Laboratory which I hope will provide a sound basis for follow on implementation efforts to fully implement this data base system.

I would like to express my deep appreciation to Dr. Thomas C. Hartrum, who as my research advisor gave me valuable guidance and encouragement. Also, I thank my thesis readers, Dr. Gary B. Lamont, Major Charles W. Lillie and Captain Richard L. Conn whose constructive comments helped to improve the clarity of this thesis. In this vein, thanks are also due to Dan Zambon, an AFIT/ENE technician, for his many hours of help with the computer systems I used.

Finally, I wish to thank my wife, Carol, for her help and encouragement during these past 18 months.

Robert W. Fonden

Contents

Page

Preface. ii

List of Figures. vii

Abstract ix

I. Introduction 1

Background 1
Statement of the Problem 4
Scope. 5
Approach 5
Overview of the Thesis 7

II. Background 9

Introduction 9
Backend Data Base Computer Systems 10

Concepts 10
Advantages and Disadvantages 14

Performance. 14
Additional Resources 16
Specialization 16
Modularity 17
Security and Integrity 17
Reliability 18
Cost 19

Summary. 19

The Relational Data Model. 20

Data Base Computer Architectures 39

Categorization 39
Single Processor Indirect Search 41
Single Processor Direct Search 44

CAFS Architecture 45

Multiple Processor Direct Search 48

CASSM Architecture 50

Multiple Processor Indirect Search 54

RAP Architecture 56
DIRECT Architecture 59

Multiple Processor Combined Search 61

DBC Architecture 61

Comparisons 64

Summary. 67

III. System Development 68

Introduction 68
Longterm Requirements and Goals. 68
System Design. 73

General Description. 73
System Functions 80
System Hardware Architecture 83
System Support Software. 85

System Development Plan. 86

General Description. 86
Development Stages 87
Resource Requirements. 89

Summary. 91

IV. Subsystem Design 93

Introduction 93
Query Operation Formats 93
Subsystem Data Tables 94

Domain Name Table 95
Attribute Name Table 95
Relation Table 95
Relation Attribute Table 96
Relation Page Table 96
Query Processor Table 97
Intermediate Memory Module Table 97
Query Packet Wait Queue 98
Query Packet Operation List 98
Query Packet Execution List 98
Query Operation State Table 99
Transmit Queue 99

Subsystem Functions. 100

Initialization 100
Setup 102
CheckQueues 103
Service Host Request 104
Assign a Query Processor 105

Service a Query Processor 106
Answer a Host Request 107
Backup the Data Base 108
Wrapup 108

Subsystem Logic Flow 108
Interprocessor Communication 117

Host / BCP Communication 118
BCP / QP Communication 118

Summary 119

V. Subsystem Implementation and Testing 121

Introduction 121
Implementation and Test Approach 122
Resources and Constraints 125
Test Support Software. 127

Host Processor Software 128
Query Processor Software 128
IMM Processor Software 129

Test Results and Evaluation. 129

VI. Conclusions and Recommendations. 131

Overview 131
Recommendations 132
Final Comment 134

Bibliography 136

Additional Readings 138

Appendix A: BCP Structure Chart Documentation 142

Appendix B: Relational Algebra Query Operation Formats . 184

Appendix C: BCP System Table Formats 196

Appendix D: BCP System Data Base File Formats 204

Appendix E: Inter-processor Architectural Specs. 210

Appendix F: Channel Link Software Specifications 212

Appendix G: HOST Structure Chart Documentation 222

Appendix H: QP Structure Chart Documentation 231

Volume II: Program Documentation (Available from AFIT/EN)

Vita 239

List of Figures

Figure		Page
1	Backend Data Base Computer System	11
2	Software Organization of Backend DBC System . .	13
3	The Relation COURSE	21
4	Levels of Normalization	24
5	A "Selection" on the Relation COURSE	27
6	A "Projection" on the Relation COURSE	28
7	The Relation QNS	30
8	A "Join" of the Relations QNS and COURSE . . .	31
9	The Medical Database	35
10	The Relational Algebra Hierarchy for a Query .	36
11	The MADMAN System Architecture	43
12	The CAFS Incorporated Retrieval System	46
13	The CAFS Architecture	47
14	The MPDS ARchitecture	49
15	The CASSM System Architecture	52
16	The CASSM Query Processor Architecture	53
17	The MPIS Architecture	55
18	The RAP System Architecture	57
19	The RAP Query Processor Architecture	58
20	The DIRECT System Architecture	60
21	The DBC System Architecture	63
22	The Conceptual Design Approach	74
23	The Physical Design Approach	75
24	The BCP Subsystem Functions	101

Figure Page

25 The BCP Test Hardware Configuration 126

26 The Inter-processor Architectural Specs 211

Abstract

A backend multiple-processor relational data base computer system was designed with the goal of implementing a data base management system using state-of-the-art technology. The objective was to overcome the traditional limitations of data base management systems implemented on conventional type computer architectures. Hopefully this would solve the ever-growing problem of information systems becoming obsolete in supporting the growing information needs of the corporate industry.

Toward this goal, investigations were made into studies in the literature involving backend data base computer systems, the relational data model, and data base computers using specialized architectures. The advantages and disadvantages of these three areas were explored and then, after having defined the longterm requirements and goals for the development of such a system, the beneficial characteristics from each of these areas were merged together to produce a system design. Central to this design is the use of a set of processors, managed by a backend controller processor, to take full advantage of three levels of parallelism in processing relational algebra query requests against relations.

Due to the complexity and size of this development effort, a top-down structured detailed design and only a partial implementation of the backend controller processor was achieved in this research effort. A detailed

development plan has been defined, consisting of several projected follow-on development efforts, to complete the entire development of this data base computer system.

I. INTRODUCTION

Background

In recent years, a substantial increase in the amount of information processing involving the use of database management systems (DBMS) has occurred. There has been a growing need for the availability of very large amounts of information as well as an equally growing need to acquire and access such information at a much faster rate. In many situations, due to this increase in demand, installations have reached the point of resource saturation and system degradation. Those installations supporting such DBMSs are developing into very serious bottlenecks in the overall function of the organizations they were built to serve. The faltering performance of these installations must be remedied, and remedied to a level of sufficiency so as to remain a suitable service for many years to come. The approach to rectifying this problem consists of two main endeavors. The first deals with the improvement of computer software and the second deals with the improvement of computer hardware.

In the area of computer software, effort is being spent on the improvement of DBMS designs. Research in the area of algorithm and file structure improvements are being conducted to increase the level of efficiency, and therefore, to speed up processing. Related to this research are studies being done to optimize the performance of information query processing. Of the three major DBMS design approaches (hierarchical, network, and relational), the relational approach has lately received an extra amount of attention. Many advantages exist in the relational approach as compared to both the hierarchical and network approaches, making the relational approach a very promising DBMS for the future. Some of the most relevant advantages are listed below:

(1) it provides a simpler, more unified user data model, resulting in systems that are easier to use and maintain;

(2) it is much more data-independent and consequently results in systems that are generalized, having databases that are easier to alter, e.g., when new data relationships are discovered;

(3) it is much easier to express data semantic integrity constraints;

(4) data retrieval and modification requests are easier to express;

(5) since information is represented in one and only one way in a relational database, only one operator is needed for each of the basic functions to perform;

(6) the emphasis is on the use of sets, rather than on the handling of one record at a time;

(7) sharing and protection requirements are more easily satisfied, due primarily to the simplicity of the underlying data base model and absence of highly

distributed access paths; and

(8) implementation issues are isolated from the logical database model - this results in increased intersystem compatibility and, most significantly, encourages a structured approach to implementation.

In the area of computer hardware, until recently, the obvious solution to the problem of system saturation was to upgrade the mainframe. This is now becoming an increasingly unappealing and/or insufficient solution to the problem. Upgrading to a new machine is not only very expensive, dollar wise; it may also require an extensive manpower expenditure to have programs and data transferred and possibly modified. One alternative to such an upgrade is the offloading of the data base management functions from the existing mainframe computer to an attached mini-computer. Such a configuration is known as a frontend/backend DBM computer system. Such an alternative can be up to an order of magnitude cheaper, in terms of hardware costs, than replacing the mainframe with a larger machine. Added to that is the fact that only the DBMS software need be moved over. In its most elementary form, a frontend/backend DBM computer system consists of a locally connected pair of computers. The application programs are executed by the frontend (Host computer), while the backend (DBMS) computer controls access to the database. The addition of the backend computer, if using an operating system supporting multiprogramming, can go as far as permitting the application program execution, the DBMS execution, and the secondary storage access, all to occur simultaneously. More complex frontend/backend DBMS forms consist of having multiple backend processors in a configuration tied to a host or even multiple host computers. Also, a number of multiple-processor backend DBMS configurations are being considered, such as having a set of processors distributed by data, distributed by function, or pipelined.

To date, relational DBMSs in general have been criticized because of their inefficiency and comparative slowness when actually implemented. But the positive attributes characteristic in a relational DBMS clearly show that this is still the best approach to take. Therefore, it is clearly a must to continue to improve on these few unfavorable qualities.

By seeking the best of two worlds, the use of backend multiple-processor computer architectures and relational DBMS concepts, a longterm solution to this information processing problem can definitely be reached.

Statement of the Problem

The purpose of this thesis is to solve four problems. First is to determine the feasibility of applying multiple-processor techniques to the implementation of a relational DBMS within a mini/micro-computer system environment. Second is to determine the system requirements, to include both long-term requirements and goals as well as short-term goals (thesis timeframe). Third is to develop a structured, modular system design (software and hardware using current state-of-the-art technology) for implementing the required system over the longterm, identifying experimental tradeoffs. Fourth is to implement a first stage system model to show feasibility and to investigate tradeoff alternatives.

Scope

The original scope of this thesis was to fully accomplish the four problems/goals defined within this thesis. After the design phase was completed, it became apparent that the implementation and testing phase would exceed the available time. Therefore, it was decided to pursue the implementation and testing of the software in a top-down manner, completing entire functions at a time, for as far as time would permit. However, due to the top-down modular design of the subsystem, and the fact that all of the software is already in clean compile form, implementation and testing of the remaining untested software should be an easy task to accomplish.

Approach

The first step consisted of an extensive literature search to determine what research and development had already been done in the area of frontend/backend DBMS configuration approaches as well as in the area of relational DBMS parallel processing approaches. An effort was made to collect information from all areas pertaining to backend and relational database systems: backend computer architectures, data base machines, state-of-the-art memory technology, relational DBMSs, storage structures, query processing, and interprocessor coupling (communication). Existing backend data base machine architectures were studied in particular.

Once a sufficient background had been gained, a feasibility study of the application of current frontend/backend system configuration concepts to a mini/micro-computer and relational DBMS environment was subsequently made.

Upon completion of the feasibility study, a requirements analysis was then accomplished. Longterm requirements and goals were identified in preparation for the development of a backend relational DBM computer system. These longterm requirements and goals were specified in detail to ensure that the system would be developed using the latest state-of-the-art hardware and software capabilities.

An approach to achieving these longterm requirements and goals was then developed. First, an initial general design of the backend system was accomplished. An investigation of design alternatives for each of the sub-systems of the general design was then performed. Choices were then made in specifically how each subsection's design would be developed. Then the system functions and interrelationships, the hardware architecture, the support software, the overall system logic flow, and the experimental tradeoffs were defined. Upon completion of the systems design, a system development plan was developed. Specific development stages were identified, specifying exactly what capabilities would be developed at each stage. Stages which could be accomplished in parallel were then identified. Finally, the resources required for each stage of development were specified so as to enable the future scheduling of the procurement and/or development of such required resources.

Once the longterm system was identified and its development approach specified, the backend controller processing subsystem was selected and pursued in greater detail - its analysis, detailed design, and implementation. A top-down modular design was used to allow for a straightforward implementation of this thesis effort as well as to provide for future ease in testing alternative subsection designs.

Overview of the Thesis

The structure of this thesis basically follows the approach that was taken in the investigation. Chapter II presents a background review of the concepts for the following: (1) Backend data base computer systems, (2) The relational data model and, (3) Data base computer architectures. In Chapter III, the system requirements are first identified, then the general design is presented, and finially the development plan is presented. Chapter IV describes the detailed design of the master-control subsystem of the backend computer system. Chapter V describes the entire effort of the subsystem's implementation and testing. Finally, Chapter VI summarizes this investigation and gives recommendations for the follow-on thesis research efforts.

II. BACKGROUND

Introduction

To ensure that the reader will understand the reasoning process behind the numerous decisions made in the architectural design of the backend multiple-processor relational data base computer system developed in this thesis effort, a background chapter has been included in this thesis. Three major sections are presented in this background chapter. Due to the importance of the many factors which all contribute to the overall design justification of a backend multiple-processor relational data base computer system, a degree of detail will be given in each of these sections.

In the first section, backend data base computer systems are discussed. The general concepts and the advantages and disadvantages of using a backend database computer system are presented to show the initial potential of using such a system in the field of information management.

In the second section, the relational data model is discussed. A detailed description is first given and then the advantages in using the relational data model for implementating a DBMS within a backend computer system are given. It is important that the reader firmly grasp the information in this section before proceeding on to the next section of this background chapter or, for that matter, all remaining chapters in this thesis.

The third and last section of this background chapter presents the details of database computer architectures built especially for the implementation of relational DBMSs using specialized state-of-the-art technology. Specific implementations are presented identifying their design specifications and their advantages and disadvantages of such architectures. Again, the level of detail is felt to be a necessity in order to properly orient the reader for the following chapters presenting the design approach chosen for this thesis effort.

Backend Data Base Computer Systems

Concepts. The basic concept of the backend data base computer (DBC) system is to remove all or part of the DBMS from the host and put it on a backend computer system (Maryanski, 1980: 3). In principle, this backend DBC may be either a general- purpose computer, for example, a mini/micro-computer, or a special-purpose computer designed specifically for data base management. A backend DBC system, in its most elementary form, is shown in Figure 1. The application programs are executed by the host computer, while the backend machine controls access to the database.

In this thesis, the term "backend DBC system" is used instead of the term "backend data base processor" because more than just a single processor may be involved. Modified mass storage devices and controllers, parallel processors,

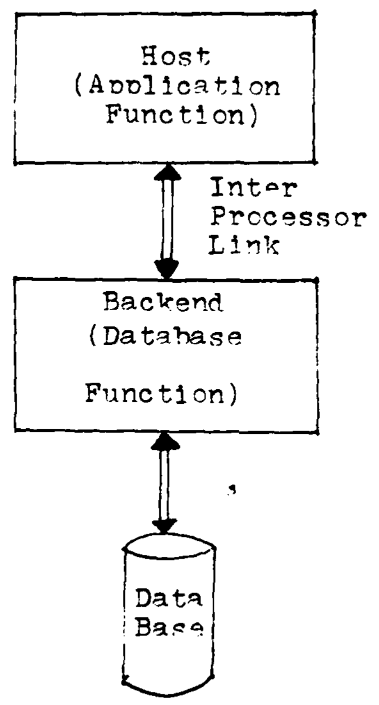


Figure 1 Backend Data Base Computer System
(Maryanski, 1980:4)

and memory devices may all be included in what is actually a computer system for data management.

In this frontend/backend DBC system, there is relatively tight coupling between the host and backend. This is essentially a master-slave configuration in which the backend database system fullfills the data requests passed to it from the host. The backend system can reject requests, however, which do not conform to the DBMS access controls. The connection between the host and the backend is usually through an I/O channel. This link should be of at least channel speed if interprocessor communication is not to become a bottleneck. Simulation studies have shown that intermachine links with speeds of 1 Mbaud or greater will produce only minimal communication delays (Canaday, 1974; Maryanski 1976). Configurations with the host and backend computers sharing common memory are recommended for situations in which quick response is absolutely necessary.

In order to form a backend DBC system, communication and interface software modules must be added to each computer. Figure 2 illustrates the software organization of a basic backend DBC system. The communication software must provide the facilities for the transmission of commands, data, and status information between the host and the backend machines. The interface routines are the processes that exchange the information via the communication system.

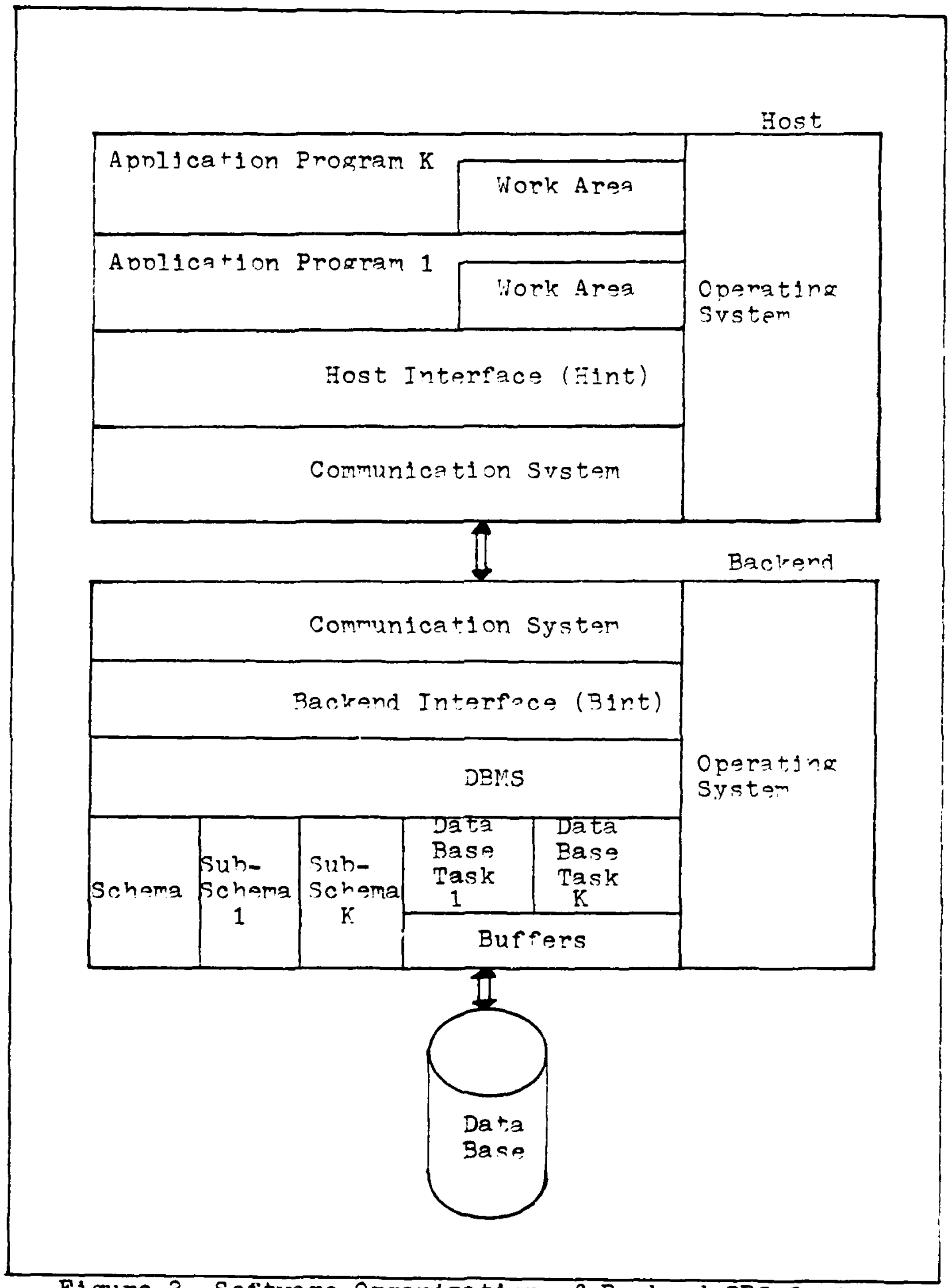


Figure 2 Software Organization of Backend DBS System (Maryanski, 1980:6)

Figure 2 shows a number of database tasks (query requests) residing in the backend machine. If a backend is to operate efficiently, the DBMS residing within the backend should function in at least a multiprogrammed environment, if only a single processor exists in the backend computer (Canaday, 1974; Maryanski, 1976).

The major architectural distinction between backend DBC systems is whether the implementation is on a general-purpose single processor computer or a special-purpose multiple processor computer.

One common assumption is that the DBMS functions are I/O bound and therefore could be performed on a less powerful processor, such as a mini-computer, more efficiently that by a large general-purpose host.

Advantages and Disadvantages. Backend DBC systems have received considerable attention due to their potential for improving the performance of installations with heavy database requirements. The research into these systems has identified several important areas in which the utilization of backend machines may provide considerable advantages. However, there are also potential drawbacks and problems that can arise if the backend concept is not properly realized.

Performance. The prime benefit of a backend DBC system is performance. This is obtained through concurrent processing, associative addressing, dedicated hardware, faster components, and other techniques applied to the data base management task.

By just moving the DBMS over to a backend computer, concurrent operation of the host (application software execution) with the backend computer (DBMS software execution) and the secondary storage devices (secondary storage access) is achieved. Further concurrent processing can then be achieved if the backend computer is composed of a multiple number of processors.

By using associative (content-addressable) storage, data can be addressed/searched in parallel, thereby further improving the performance of the system.

Performance is also improved by reducing the volume of data that must be moved between the host and the data storage devices. This can be accomplished by using the relational operations 'selection' and `projection'; and data compression or encoding. Using selection, the backend computer sends only those records actually needed by the application program. In a typical case in today's standard DBMSs, a thousand records may be read to find the few that answer the user's query. Projection can then be used to reduce the data volume for those cases where only a few fields in the record are needed. In this case, the backend computer selects only the desired records and the extracts (projects) and converts only the required fields.

By removing the DBMS from the host, the overhead of interrupt handling of I/O generated by the DBMS is also removed from the host, further improving the system's performance.

It must be emphasized that in order to achieve a real performance improvement, a highspeed link between the host and backend must be used, and a substantial demand for database access must exist. If the gains due to concurrency do not outweigh the losses caused by communication overhead (interface and communication software, and the transmission time of the interprocessor link), then the move to a backend DBC system would be a fruitless effort.

Additional Resources. By off-loading part of the processing from the host to the backend computer, additional processing time and memory on the host is released and can be used by the application programs. This benefit alone can be critical if the host is approaching saturation. If the interface and communication software do not require all of the released space, then additional application programs can reside on the host machine, resulting in an increase in concurrency and throughput.

Specialization. Specialization also provides certain benefits in the hardware area. If special-purpose hardware is used for the backend computer, certain features and their corresponding costs are eliminated. For example, a backend computer does not need such features as fast multiply and divide logic, floating-point hardware, and the long-word-size registers necessary to obtain high precision in complex arithmetic calculations. It does, however, need a powerful set of byte manipulation instructions and a high-speed I/O capability to a wide variety of storage devices.

Modularity. The principle of modular design is generally applied in the creation of both hardware and software in order to produce well defined, reliable, and correct devices and programs. A backend DBC system is an example of modular design at the computer system level. This seperation of functions provides for allowing multiple hosts to interface with a single data base or distributed database network. Once the data base and the corresponding DBMS are on the backend computer, the data base can then be made available to other types of host computers by providing an interface so that the new host can communicate with and receive data from the backend computer. This is a much simpler task than modifying the entire DBMS or developing methods to allow two independent DBMSs to concurrently access the same data base.

Security and Integrity. Another backend DBC system benefit is the improvement of the security and integrity of the data base. Because both are related to access control, security and integrity are considered together. A backend DBC system provides a single access path to the data base. If the data base devices are connected only to the backend computer, then any operation on the data base must go through the backend computer. Since the backend computer is a system resource and therefore is not programmable by the applications programmers, it is now impossible for an application program to circumvent the DBMS and access the data files directly. This effectively isolates the application and the host from the data base and thus increases its security and integrity. In order to gain unauthorized access to data in a backend DBMS, the intruder must appear to the DBMS as a valid user. Thus the backend DBMS causes the intruder to concentrate penetration efforts on defeating the password mechanism or monitoring the behavior of the communication interface, a much more difficult and lengthy effort.

Reliability. Improved backup and recovery is another possible benefit of a backend DBC system. With a separate host and backend computer, the two machines can provide a check on each other. In fact, Canaday proposes two separate logs for recovery (Canaday, 1974). In one, the host records all transactions or requests sent to the backend computer. Thus if the backend computer fails, the data base can be restored using this log. Similarly, the backend computer would keep a record of all changes to the database. If the host fails, then the recovery can be performed using the backend computer's log. In each case, the recovery is done with an audit trail from the machine that did not fail in order to minimize the chance that the audit trail was contaminated. Moreover, by having one processor check on the other, failures can be detected sooner. Thus there will be less chance of the resulting errors being propogated to other parts of the data base.

Cost. As mentioned earlier, a primary motivation for the backend DBMS work is the development of an economic alternative to the upgrading of a large mainframe. The attachment of a dedicated conventional mini-computer to an existing mainframe is an order of magnitude cheaper, in terms of hardware costs, than replacing the mainframe with a larger machine. But other factors must be considered in the evaluation of the economic alternatives. The media used for the storage of the database on the existing mainframe and the backend machine must be compatible in order to avoid a substantial conversion cost. Backend DBMS communication and interface software must also be included in the pricing. When all factors are considered, the great difference in price between that of a mainframe to replace the entire existing system and of a minicomputer to assume the database function is the factor that makes a backend DBMS a viable economic alternative to a mainframe upgrade.

Summary. In summary, the cost/performance benefits of a backend DBC system and the increased functional capabilities that it may offer are its prime advantages. The backend computer can be used to extend the useful life of an existing general-purpose host and to reduce the costs and complexity of the conversion when the host is replaced. Finally, a backend computer makes it easier and cheaper to integrate a heterogeneous network into an existing data base system. Potential trouble spots include multivendor coordination in dealing with ambiguous failures of hardware and software, and the proper sizing of the backend computer to ensure that the host-backend system remains balanced in handling a system load having projected growth.

The Relational Data Model

A relation has a precise mathematical definition.

Definition 2.1: Given sets S1, S2, ..., Sn (not necessarily distinct), R is a relation if it is a subset of the Cartesian product S1 x S2 x ... x Sn. That is, R is a set of ordered tuples s1, s2, ..., sn such that s1 belongs to S1, s2 belongs to S2, ..., and sn belongs to Sn. The sets S1, S2, ..., Sn are called domains of R; the value of n is called the degree of R (Date, 1977: 73).

A relation, as shown in Figure 3, is a two dimensional table with the following properties: (1) no two rows are identical (2) the ordering of the rows is insignificant, (3) the ordering of the columns is insignificant, and, (4) all table entries are atomic (nondecomposable) data items.

Each row of the relation is called a tuple. If a relation has n columns (i.e., degree n), each row is referred to as an n-tuple. For example, each tuple of Figure 3 is referred to as a 6- tuple.

Each column is called an attribute; each attribute has a domain. The domain is the set of values which can appear in that column. For example, in Figure 3, the domain of the attribute NUMBER is EE550, EE646, EE692, MA580, MA531,

Number	Title	CR Hours	L Hours	LB Hours	Size Limit
EE550	Intro To Logic Design	5	4	2	35
EE646	Data Base Systems	4	4	0	25
EE692	Advanced Computer Architecture	4	4	0	30
MA580	Probability and Statistics	3	3	0	50
MA531	Math Methods Of Computer Science	4	4	0	50
EE799	Thesis	4	4	0	99
MA445	Intro To Algorithm Design	4	4	0	20
EE690	Digital Language Laboratory	2	2	2	10
EE545	Software Systems Acquistion	4	4	0	75
MA799	Thesis	4	4	0	99

Figure 3 The Relation Course (Date, 1977:154)

EE799, and others.

A relation resembles a file, a tuple a record (occurance, not type), and an attribute a field (type, not occurance). A relational system contains several different relations just as a CODASYL system contains several different files. Relations may also be used to indicate a relationship between different types of entities in addition to describing a particular entity type. For example, a data base may contain a COURSE relation and a QUARTER relation as well as a explicit COURSE-QUARTER relation linking the two types of entities. Different relations are linked together or related through common attributes.

An attribute (or combination of two or more attributes) whose domain contains values which uniquely identify the n- tuples of the relation is called the primary key. For example, in Figure 3, the primary key is NUMBER. A primary key is said to be nonredundant if it is either a simple domain (not a combination) or a combination such that none of its constituents is superfluous in uniquely identifying the tuple. For example, in Figure 3, (NUMBER,TITLE) would not be a nonredundant primary key for COURSE. A relation can contain more than one nonredundant primary key; in that case, one of these candidates is arbitrarily selected and called the primary key of that relation.

A normalized relation is one in which each attribute contains only atomic (nondecomposable) values. Three levels of normalization have been defined. All normalized relations are in first normal form; some of the first normal form relations are also in second normal form; and some of the second normal form relations are also in third normal form. See Figure 4.

In order to discuss the differences in these three forms, the terms "functional dependence" and "full functional dependence" must be defined.

Definition 2.2: Given a relation R, the attribute Y of R is functionally dependent on attribute X of R if and only if each X- value in R has associated with it precisely one Y- value in R at any one time (Date, 1977: 154).

For example, in Figure 3, TITLE, CRHOURS, LHOURS, LBHOURS and SIZELIMIT are functionally dependent on NUMBER; that is, given a particular NUMBER value, there exists precisely one corresponding value for each of TITLE, CRHOURS, LHOURS, LBHOURS and SIZELIMIT. Note that functional dependence can be applied when either X or Y or both are composite domains.

Definition 2.3: Attribute Y is fully functionally dependent on attribute X if it is functionally dependent on X and not functionally dependent on any subset of the attributes of X (X must be composite) (Date, 1977: 155).

For example, in Figure 3, the attribute CRHOURS is functionally dependent on the composite attribute (NUMBER,TITLE), but is not fully functionally dependent on

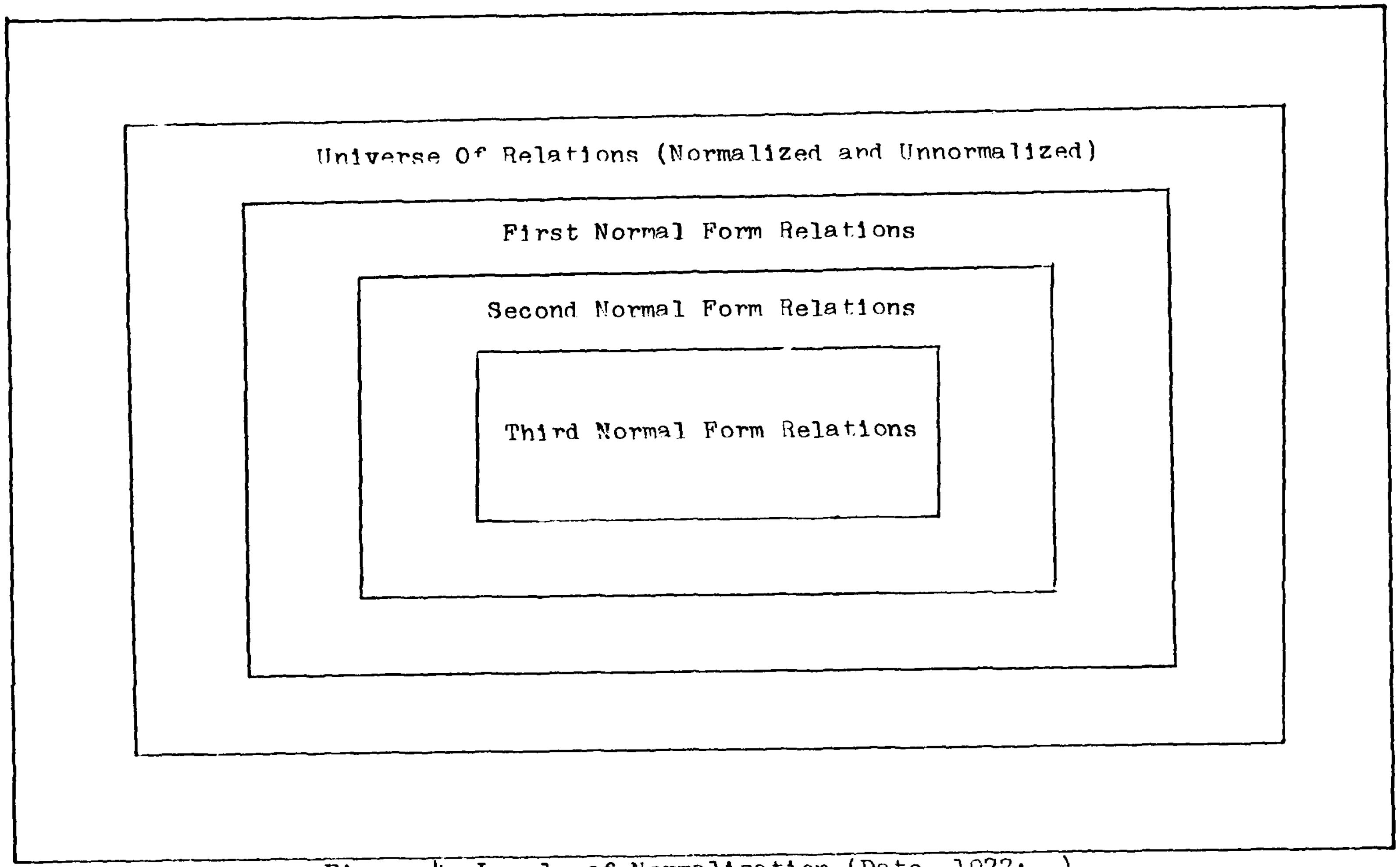

Universe Of Relations (Normalized and Unnormalized)
First Normal Form Relations
Second Normal Form Relations
Third Normal Form Relations

Figure 4 Levels of Normalization (Date, 1977:)

(NUMBER, TITLE) since it is functionally dependent on the subset NUMBER.

With the above definitions in mind, three normalized relation forms can now be defined.

Definition 2.4: A relation R is in first normal form if and only if all underlying attributes contain atomic values only. A relation R is in second normal form if it is in first normal form and every non-key attribute is fully dependent on the primary key. A relation R is in third normal form if it is in second normal form and every non-key attribute is nontransitively dependent on the primary key (Date, 1977: 157).

By restricting relations to third normal form, relational systems can eliminate the occurance of redundancy and update anomalies. Thus, when a tuple is modified, added, or deleted, normalization will ensure that changes are made only to that single tuple.

Operations on a relational database may be specified in either a relational algebra or a relational calculus, corresponding to a low- or high-level query language. The relational calculus specifies the desired output of the query and allows the DBMS to select the appropriate method to obtain the results. A relational algebra must specify not only the output, but also the method (steps) to obtain it.

Relational operations may be classified by their handling of a single relation or by more complex treatment of multiple relations. The basic operations on a single relation include selection, projection, modification, addition, and deletion. Selection identifies the desired tuples in a relation by specifying the values of certain attributes within the relation. If a value of the primary key domain is specified, only a single record or no records are selected. If the primary key is not one of the attributes specified, however, many records may be selected. For example, Figure 5 is a selection result of "List COURSES where CRHOURS equals 4."

Projection identifies the attributes of interest within the selected tuples. When multiple records are selected, the elimination of some attributes by the projection operation may leave duplicate entries among the partial tuples that remain. Therefore, there are two types of projection - one in which the duplicates have been eliminated, and one in which they are included. Figure 6 shows a projection of the relation COURSE (Figure 3) on the attributes TITLE and CRHOURS. Note that only 9 tuples appear in the projection; the tenth tuple, THESIS, has been omitted since it is a duplicate of the sixth tuple.

A modification operation changes the value of an attribute in an existing tuple. Additions and deletions are relatively simple because the relation's tuples are of a fixed length and are not maintained in any (sorted) order.

Division, intersection, union, and difference are four

Number	Title	CR Hours	L Hours	LB Hours	Size Limit
EE646	Data Base Systems	4	4	0	35
EE692	Advanced Computer Architecture	4	4	0	30
MA531	Math Methods Of Computer Science	4	4	0	50
EE799	Thesis	4	4	0	99
MA445	Intro To Algorithm Design	4	4	0	20
EE445	Software Systems Acquisition	4	4	0	75
MA799	Thesis	4	0	0	99

Figure 5 A "Selection " On The Relation Course

Title	CR Hours
Intro To Logic Design	5
Data Base Systems	4
Advanced Computer Architecture	4
Probability and Statistics	3
Math Methods Of Computer Science	4
Thesis	4
Intro To Algorithm Design	4
Digital Language Laboratory	2
Software Systems Acquisition	4

Figure 6 A "Projection" On The Relation Course

other operations that are sometimes included in relational data base processing. Division is a binary operation on two relations that results in a new relation. It is useful for answering such queries as "List all QUARTERS that offer both COURSE MA746 and COURSE EE692." The remaining operations are set theory operations that also operate on two relations to produce a third. In each case, the relations used as operands must have at least one common domain.

The primary operation on multiple relations, the "join", is the relational equivalent to the set definition in the CODASYL databases. Given two relations which have a common attribute, the join combines the tuples of each relation where the values of the common attribute are equal. The result is a third relation in which each tuple consists of a tuple from the first relation concatenated with a tuple from the second relation which contains the same attribute- value (except that one of the two identical attribute- values is eliminated). If a value in the common attribute appears in one relation and not the other, tuples containing that value do not participate in the join. For example, consider the relation QNS as shown in Figure 7. The relation QNS is joined with the relation COURSE (Figure 3) on the attribute NUMBER as shown in Figure 8. Note that several of the COURSE tuples do not appear in the join because there was no match on the NUMBER attribute value.

In the above paragraphs, the relational operations

Quarter	Number	Student
SU81	EE799	Allen
SU81	EE799	Klein
FA81	EE646	Harris
FA81	EE799	Allen
SU81	MA600	Allen
SU81	EE690	Jones
FA81	MA799	York

Figure 7 The Relation QNS

Quarter	Number	Student	Title	CR Hours	L Hours	LB Hours	Size Limit
SU81	EE799	Allen	Thesis	4	0	0	99
SU81	EE799	Klein	Thesis	4	0	0	99
SU81	MA531	Allen	Math Methods Of Computer Science	4	4	0	50
SU81	EE690	Jones	Digital Language Laboratory	2	2	2	10
FA81	EE646	Harris	Data Base Systems	4	4	0	25
FA81	EE799	Allen	Thesis	4	0	0	99
FA81	MA799	York	Thesis	4	0	0	99

Figure 8 A "Join" of the Relations QNS and Course

have been seperately discussed. However, it is also possible to select from joins or projections, join selections, join projections, project on a selection or joined relation, etc.

The relational algebra and relational calculus based retrieval languages are very powerful. They are in fact relationally complete. Relational completeness means that any derivable relation can be retrieved from the database. It means to the user that, if the information wanted is in the database, then it can be retrieved. In lower-level languages the user must write quite complicated procedures to answer all but the simplest questions.

The most obvious distinction between CODASYL and Relational systems is that CODASYL data items can be repeated but relational domains cannot. For example, a CODASYL QUARTER record may have a repeating item identifying the COURSES offered. The relational equivalent is a separate QUARTER-COURSE relation with one tuple for each QUARTER- COURSE pair. The disadvantage of this approach is that some of the data, that is, QUARTER, are repeated in each tuple. On the other hand, one advantage is that the relation's length is fixed and, therefore, usually easier to process. Relations may be thought of as highly disciplined files - the discipline concerned being one that results in a considerable simplification in the data structures the user has to deal with, and hence in a corresponding simplification in the operators needed to manipulate them.

The relational structure is very easy to understand. But simplicity of data representation is not the end of the story. Observe that the uniformity of data representation leads to a corresponding uniformity in the operator set: Since information is represented in one and only one way, only one operator is needed for each of the basic functions that one wishes to perform. This contrasts with the situation with more complex structures, where information may be represented in several ways and hence several sets of operators are required. For example, the network-based DBTG system provides two "insert" operators: STORE to create a record occurance, and CONNECT to create a link between an "owner" and a "member".

A simple query to a large relational database may take a prohibitively long time to process due to the overhead of having to use a very large pointer structure within the data base files. To decrease this time in applications requiring fast response, parallel processing is necessary. It just so happens that relational algebra queries are very conducive to being processed with a high degree of parallelism. Specifically, three levels of parallelism are capable of being performed on relational algebra queries: independent parallelism, pipelining, and node splitting. Under certain conditions, the exploitation of these levels makes highly parallel processing and short response times possible.

Independent parallelism includes the parallelism between separate queries and the parallelism between independent operations in a single query. Pipelining parallelism includes the parallelism between succeeding operations of a relational algebra query applied to different portions of data. Node splitting parallelism includes the parallelism between duplicate copies of an operation in a relational algebra query applied to different portions of a relation's data.

For example, consider a query of the following database of three relations shown in Figure 9. R1 and R2 contains information about hospitals and doctors respectively. R3 contains information about the "employment" relationship between hospitals and doctors (including the distance between a hospital and the residences of the doctors it employs). The query is "Find all the hospitals in New York city which have emergency treatment facilities (STATUS > 20) and heart specialists living within 30 miles." The relational algebra query for the above consists of the following steps.

S1: Select R1 where CITY = 'NY' ^ STATUS > 20 giving TR1.

S2: Select R2 where SPECIALTY = 'HEART' giving TR2.

S3: Select R3 where DISTANCE < 30 giving TR3.

J1: Join TR1 and TR3 over H# giving TR4.

J2: Join TR2 and TR4 over D# giving TR5.

The relational algebra hierarchy reflecting the above steps is shown in Figure 10. First notice that all three

H#	HName	City	Status	Others

The Relation Hospital (R1)

D#	DName	Specialty	Address	Other

The Relation Doctor (R2)

D#	H#	Distance	Salary	Other

The Relation Employment (R3)

Figure 9 The Medical Database

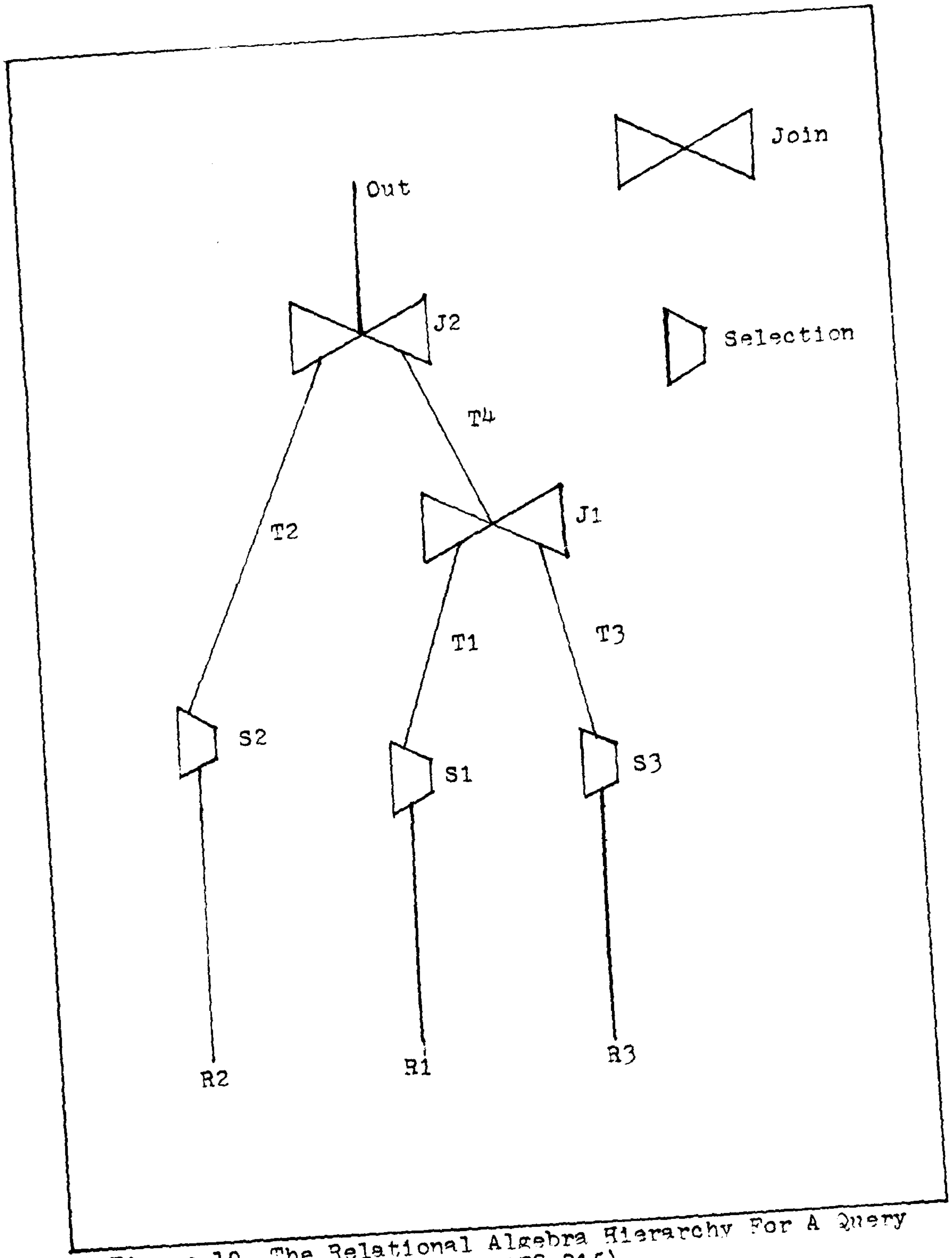


Figure 10 The Relational Algebra Hierarchy For A Query (Chang, 1978:315)

SELECTION branches S1, S2, and S3 are independent (i.e. there is no precedence relationship among them), and so they can be processed in parallel. R1, R2, and R3 have to be stored in different storage modules so that parallel access without contention is possible. The parallel processing of the three SELECTIONS can be finished in maximum of [time(S1), time(S2), time(S3)]. J1 and S2 are also independent and can be processed in parallel. The total execution time with parallel processing of the independent nodes is: time(J2) + maximum[time(S2), time(J1) + maximum<time(S1), time(S2), time(S3)>].

To further improve the response time, the inherant parallelism between adjacent nodes in the relational algebra hierarchy must be explored. Note that J1 does not have to wait for the completion of either S1 or S3 to perform its operation. J1 can use the outputs of S1 and S3, one tuple at a time, while S1 and S3 still keep producing tuples one by one. Therefore J1 and S1,S2 can be processed in parallel and form a "pipeline". The same logic can be applied with J2 and T2,T4.

Next, note that the SELECTIONS of different portions of a large relation are in fact independent processes. To further improve response time, the relations S1 and S2 can be separated into several groups of tuples and have several SELECTION (S1 and S2) processes applied to these groups concurrently. This is to "split" a node into several parallel sub-nodes. If S1 and S2 were each split into four nodes, the execution time of either process would become one-fourth of it's original time.

Response time of an urgent query can be shortened significantly by utilizing this parallelism. The identification and exploitation of parallel processable subtasks in queries has other advantages also, for example, better system utilization and simultaneous services to a large number of users.

While the logical structure and the mathematical basis of the relational data model are very appealing, it should be noted that the implementation, using a conventional Von Neumann computer, results in many of the same problems that current DBMSs try to overcome by using data models with a link structure. That is, the two dimensional table becomes a one dimensional string of data with pointers to link related data items. This results in the creation of a software interface from the logical data model to the physical storage structure. This relatively simple relational data model can be as complex as the traditional logical data models when implemented on the Von Neumann machine. This leads to the investigation of an alternative computer architecture to support the relational data model.

Data Base Computer Architectures

Categorization. As presented earlier, a lot of research has been directed toward the implementation of DBMSs on backend computer architectures. Initial work dealt with conventional backend architectures only, mostly to prove feasibility, while more recent work has dealt with the use of specialized architectures. Before discussing both in further detail, consider the following classification scheme proposed by Olin Bray (Bray, 1979a: 99). The two criteria established for this classification scheme are the number of processors involved in the data base processing and the type of hardware organization used to search for the data directly on mass storage devices or indirectly in some buffered or intermediate storage area. The five categories comprising this classification scheme are

1) Single processor indirect search (SPIS)
2) Single processor direct search (SPDS)
3) Multiple processor direct search (MPDS)
4) Multiple processor indirect search (MPIS)
5) Multiple processor combined search (MPCS)

In the indirect search approach the data is first staged from permanent storage to local memory and then searched. Implementations may use either conventional memory/storage or associative memory/storage. In the direct search approach the data is searched directly at the location where the data is stored. These implementations use only associative memory/storage.

The number of parallel processors used is important because it directly relates to performance. The types of requests that typically occur in a data base management operation involving the search of an entire file/relation or database lend themselves very well to parallel processing. In these cases, if the file/relation or database were divided into separately accessable blocks of data, a set of parallel processors could perform the search much faster than a single processor. In addition, with the cost of hardware dropping by a factor of 20 to 30 percent a year, microprocessors are a very cost-effective means of implementing parallel processing.

Where the database is searched also affects performance. The search may be performed directly on the mass storage unit where the data is permanently stored or indirectly in some intermediate storage area. The objective is to perform the search as close to the source as possible to avoid the delays due to data movement. Because of the rotational speeds and transfer speeds of mass storage devices today, however, most of the direct searches can involve only very simple selections. Complex boolean expressions just cannot be evaluated without skipping records, thus requiring one or more disk revolutions. The result is a tradeoff between query complexity and performance, and the choice between a direct-search database computer and an indirect one may depend on the applications involved.

Single Processor Indirect Search. The single processor indirect search (SPIS) corresponds to the conventional general-purpose processor. In this traditional approach, part of the database is read from its permanent storage on moving head disks into the intermediate staging storage of random access memory (RAM). Index tables and pointers are used (file management system or hierarchical DBMS or network DBMS in which data are requested by application programs a tuple-at-a-time) to determine which parts of the database are staged into the system's main memory. This block of data is then processed to determine the records that are to be retrieved or modified. Although the implementation details may differ, all current file management systems and DBMSs operate essentially in this fashion.

A backend computer architecture system would be inefficient for low-level, record-at-a- time requests because a request message and response message would have to be passed between host and backend for each and every record to be retrieved. For record-at-a-time operations, this approach would actually degrade the system's performance.

This is not true for high-level queries because the backend can relieve the host of a significant amount of work between the time it receives a request and the time it returns an answer, an answer which may consist of up tc hundreds of records.

Initial research efforts of backend computer architectures all consisted of approaches in the SPIS category. The approaches of interest were XDMS (Bell Laboratories), IDMS (Cullinane Corporation), KSU (Kansas State University), and MADMAN (General Electric). They all used mini-/micro- computers for the backend processor. The first three used very slow communication links (less than or equal to 4800 baud) while MADMAN used a shared memory link (see Figure 11). The first three used commercially developed and marketed DBMs packages loaded completely on the backend computer while the MADMAN approach developed its own DBMS package and loaded only the CPU activities of the DBMS on the backend machine, leaving the DBMS disk I/O operations on the host (see Figure 11). The main question regarding these research efforts had to do with the feasibility of efficient communication between the host software and the backend DBMS software. If the communication between the machines became a bottleneck, then such an approach would be fruitless. Such was determined not to be the case and so its feasibility was proven. A much faster communication link (approximately 1M baud) would be required, to eliminate processor delay caused by waiting on inter-processor communication transmission, but such capability is within current technology.

With the recent development of the following two

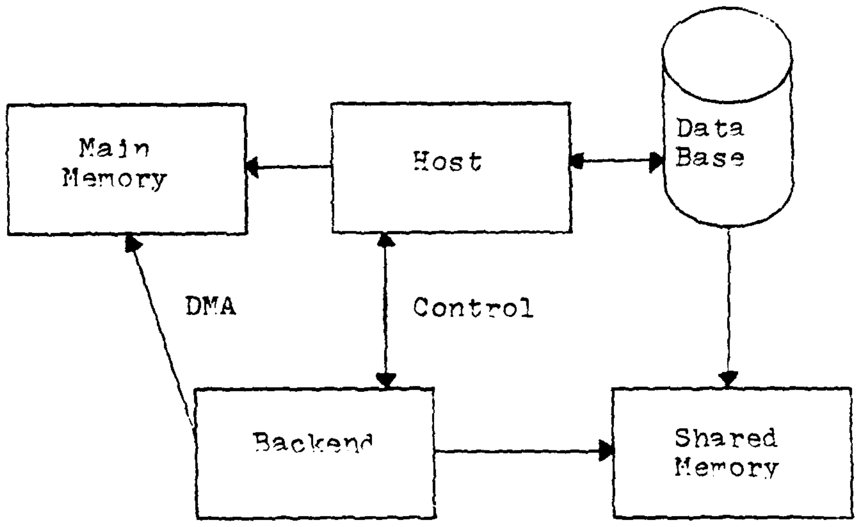


Figure 11 The Madman System Architecture
(Hutchison, 1978:85)

technologies, an ability to now quite easily take advantage of the relational model was brought about. They consist of the following.

1) Solid-state (LSI and VLSI technology) content addressable memory devices (parallel and serial) which permit the construction of longer words, minimize the cross-talk problem, require less energy and permit the use of components with loose tolerances, and

2) Intelligent memory devices such as disk, bubble and CCD having search capabilities similar to associative memory, an ability to now quite easily take advantage of the relational model, which bears a 1:1 relationship with associative storage.

The result has been new efforts which proceeded on to replace these conventional backend computers with backends using specialized architectures; the goal in sight of boosting the capability and speed of information processing systems. These special architectural approaches are described in the following discussions covering the remaining four classification categories.

<u>Single Processor Direct Search</u>. In this approach, the data are searched by the backend, and only the desired records or their specified parts are sent to the host (the relational data model is used). This reduction in data volume, using selection and projection operations, significantly reduces the work load on the host by eliminating many relatively simple tests on large amounts

of data that were normally performed by the host (tuple-at-a-time processing). This approach is classified as a direct search because intermediate storage is not used for the data search. By not having this intermediate storage, however, only a limited set of relational DBMS query operations and not a complete set can be supported.

<u>CAFS Architecture</u>. CAFS (Content Addressable File Store) was designed as a highspeed search device and is an example of the SPDS approach (Bray, 1979a: 101). Figure 12 shows the overall architecture of the system and Figure 13 shows the major components within the CAFS architecture itself. CAFS also is classified under the SISD category. Data from the disks stream through the backend where the selection is performed. Records which meet the selection criteria are then passed to the host.

Sixteen key registers are used in the comparisions as the data are streamed through the system. These results are passed to the search-evaluation unit to evaluate the complete logical expression of the query. Multiple 64K bit array buffers are incorporated in CAFS for joins and the elimination of duplicates following projections and may be used to hold input data for the search evaluation unit. The retrieval unit is used to buffer records to be sent to the host.

The essential problem that CAFS addressed was the rapid location and retrieval of records when the selection criteria are based on data values within the record rather

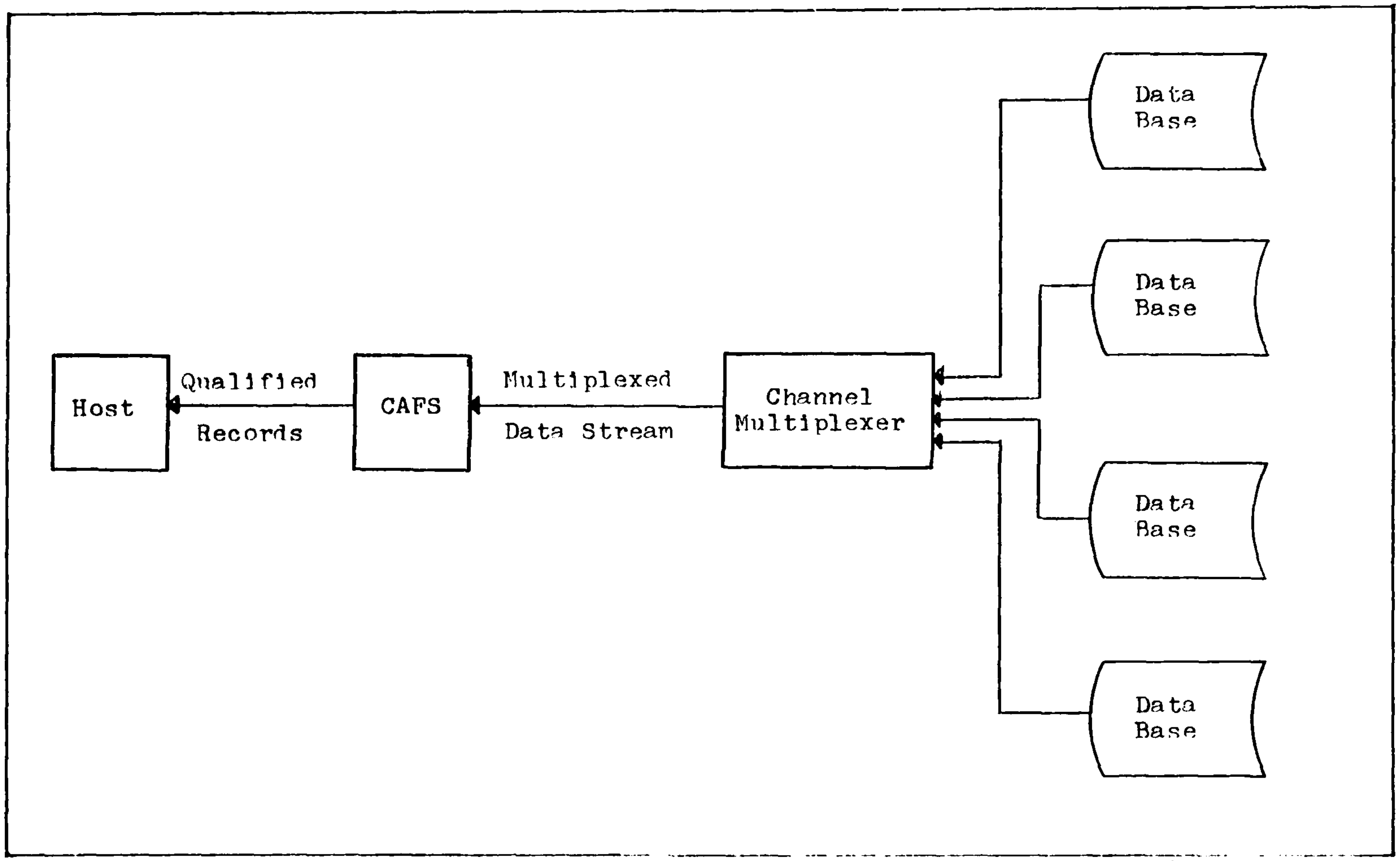


Figure 12 The CAFS Incorporated Retrieval System (Bray, 1979:102)

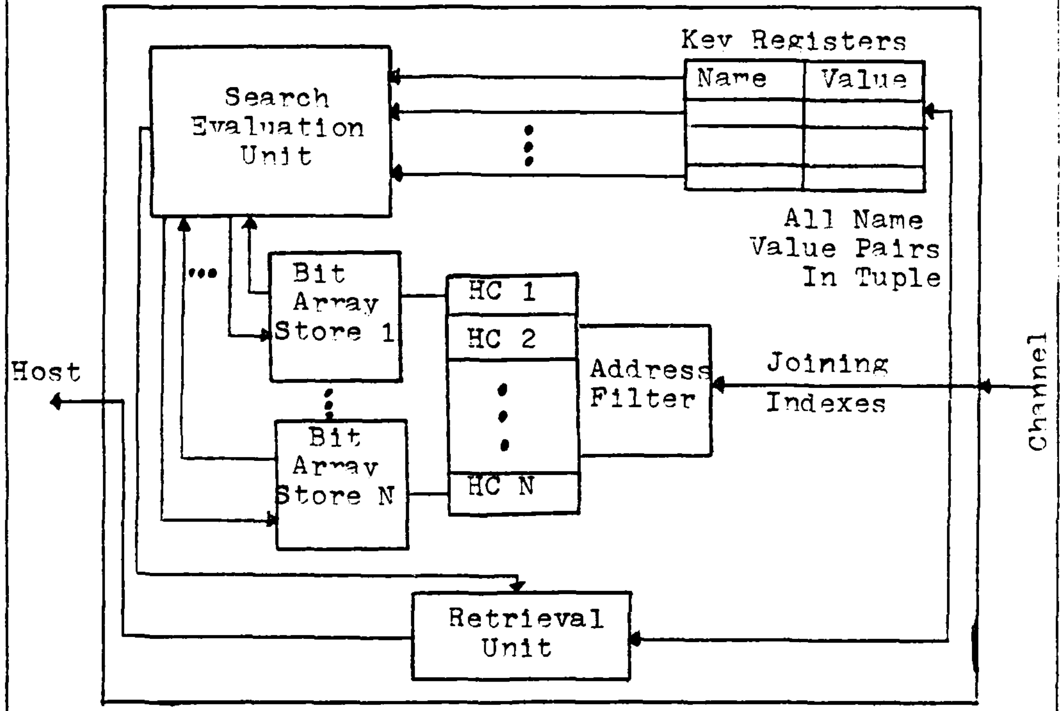


Figure 13 The CAFS Architecture (Bray, 1979:103)

than the record's position. The problem is even more severe when any of the fields in the record or parts of a field in the record can be used as search keys. CAFS was designed to overcome these problems through the use of pointers to large blocks of data, for example a track, rather than individual records. This block is then scanned serially to locate the desired records.

CAFS is strictly a retrieval system and does not allow on- line updating of data. Higher-level functions such as sum and average are left for the host to calculate. Being an SISD machine, the backend can only process one query at a time.

Multiple Processor Direct Search. Figure 14 illustrates the basic characteristics of the multiple processor direct search (MPDS) approach, of using a processor to directly search each track of the database. As the storage devices (CCDs, MBMs, drums, or head-per-track disks) rotate, data are read into the corresponding track processor, which then examines the records to determine which ones should be selected or modified. The data "stream" (transfer) off the track, through the processor and are then written back to the same track within the same revolution. The amount of processing that can be performed on a single revolution, therefore, is limited by the speed of the track processor and the speed of the rotating device. If processing is not completed in a single revolution, then flags must be set in each record

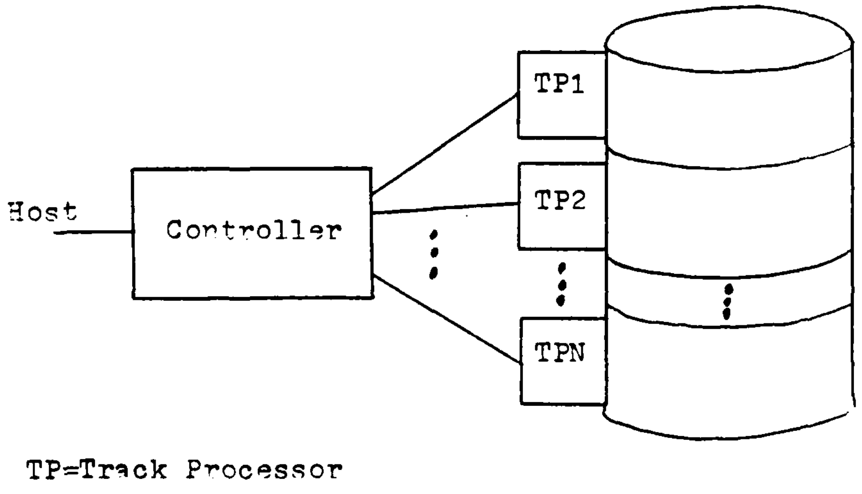


Figure 14 The Multiple Processor Direct Search
(Bray, 1979:107)

to indicate its selection and the degree of processing completed.

Using a MPDS approach, the entire database can be searched in a single revolution. In fact, by having multiple compare registers with each processor, it may be possible to answer several requests in a single revolution. Multiple flags would be needed in each record for each request being processed.

The tradeoff that has been made with the direct-search approach is to minimize the response time rather than the storage costs. Since all tracks of the database are read and processed in parallel, response time is independent of the size of the database and is simply a function of the time to read a single track. The penalty for this constant response time is the cost of storage. Storage costs grow at a rate equal to the amount of data being stored. As more tracks/disks are required, more track processors must also be added.

Unfortunately, a serious problem exists due to this continuous read from and write to the database for every revolution - backup and recovery. Constant rewriting of the entire database significantly increases the number of errors as compared to a conventional system. The ability to issue a rewrite due to a write error does not exist in this architecture.

<u>CASSM Architecture</u>. CASSM (Context Addressable Seqment Sequential Memory) was developed at the University

of Florida (Su, 1979) and is an example of the MPDS approach (see Figure 15). CASSM, a SIMD machine, performs parallel processing of the database so that the time required to perform many of the database functions is independent of the size of the database. Head-per-track disk devices were used for the database storage units. The track processors all process the same function assigned at the same time thus enabling the entire database to be processed in one revolution. Within each track processor, a set of operating modules operate in a pipelined parallel fashion in processing the instruction (see Figure 16). CASSM's design assumes that the entire database resides online and is available to all N track processors. The storage could be implemented using CCD, MBM, etc. such that the data appear to be rotating past some fixed point (bit serialized associative storage). Then, as the data rotate, every word is read, processed, and written back to this storage device. Being a SIMD architecture, CASSM only processes one query at a time.

Unlike conventional rotating storage, a CASSM track processor contains separate read and write heads for each track (see Figure 16). Every word of storage then can be read, and, if not deleted, written back on each revolution. Word insertions cause every word thereafter to be shifted down one position on the track while word deletions cause every word thereafter to be shifted forward one position on the track. CASSM was designed to be used in conjunction

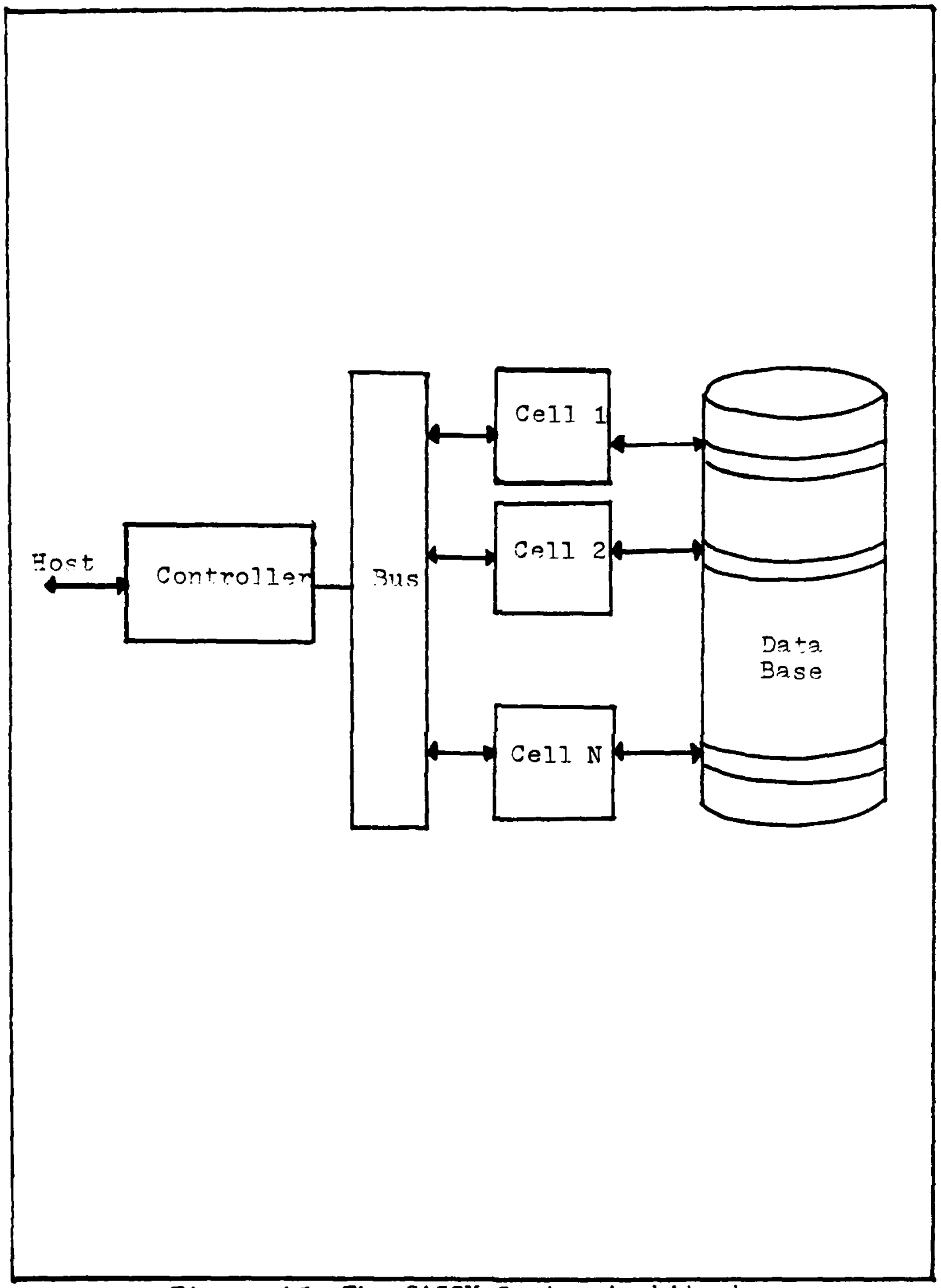


Figure 15 The CASSM System Architecture
(Bray, 1979:109)

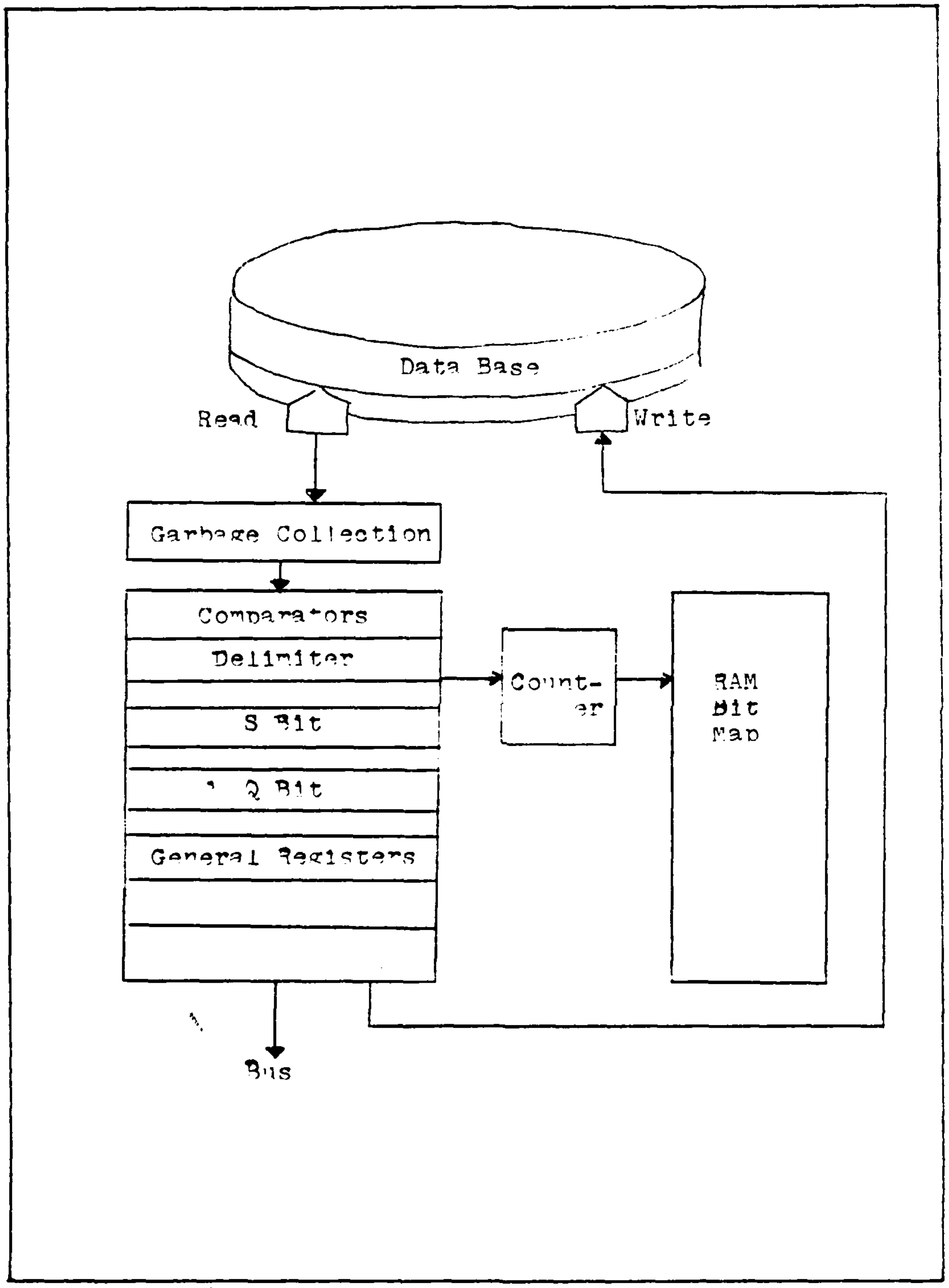


Figure 16 The CASSM Query Processor Architecture
(Bray, 1979:110)

with a general- purpose host processor issuing high- level data management queries. It was one of the first attempts to construct a special architecture for general non-numeric processing.

Multiple Processor Indirect Search. The multiple processor indirect search (MPIS) approach is very similar to the MPDS approach in that multiple processors are used to process the database in parallel. The main difference is that not all of the database is processed in parallel. Instead, a part of the database is staged into an intermediate storage device and then searched there. This approach is shown in Figure 17.

For this approach to be feasible, the capability must exist to be able to quickly identify those parts of the database which must be processed and to load them into the intermediate storage. Quick identification will still require pointers, but pointers only to large blocks of data, for example, tracks or cylinders, rather than individual records. Rapid loading of the intermediate storage will require a high data transfer rate and buffering within the intermediate storage. For the MPIS approach, the time to process a transaction is indirectly dependent on the size of the database.

The following two subsections describe two systems using the MPIS approach. RAP (Relational Associative Processor) is a SIMD type approach and DIRECT is a MIMD type approach. Both were specifically designed for non-

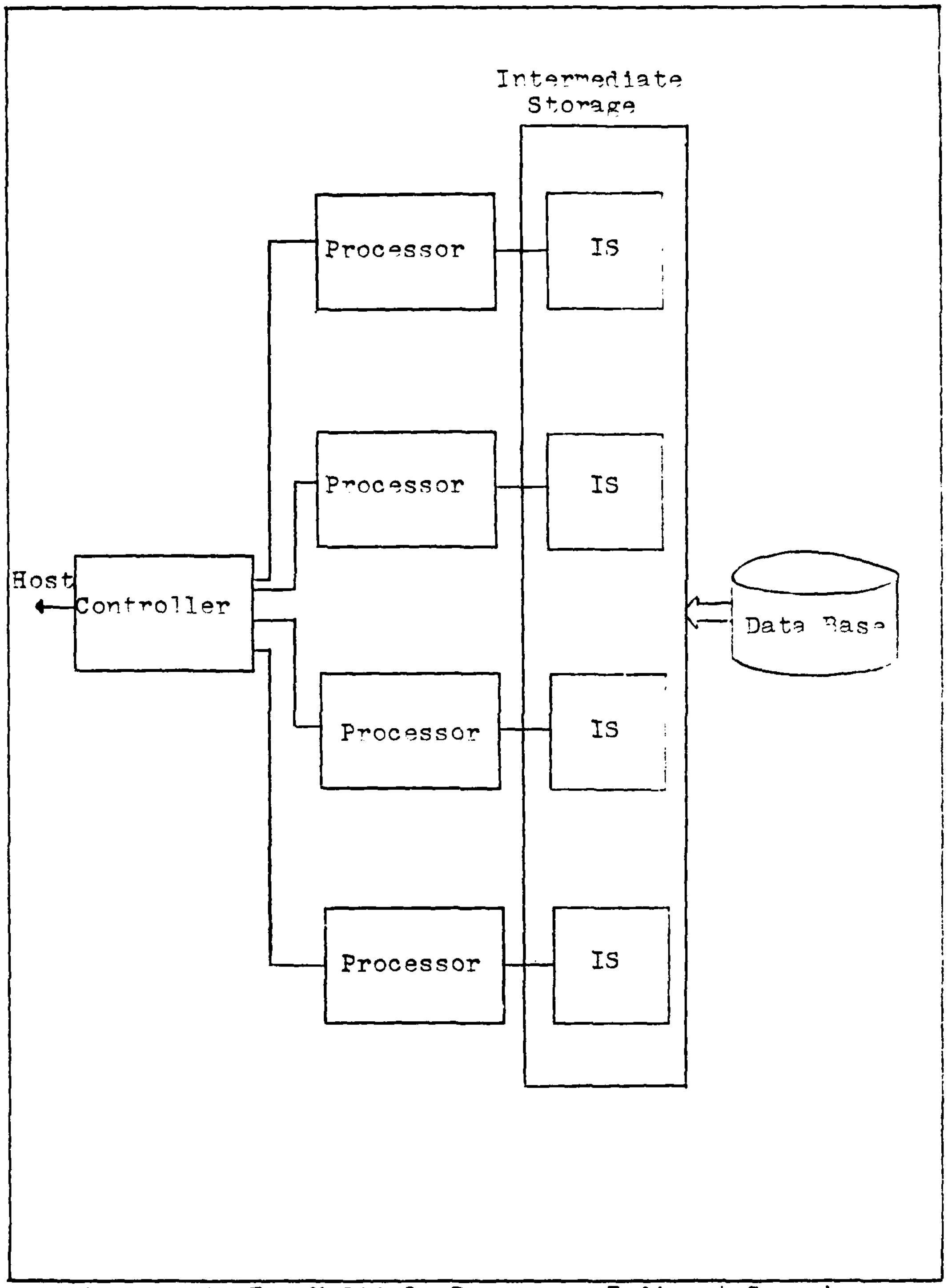


Figure 17 The Multiple Processor Indirect Search
(Bray, 1979:121)

numeric data management.

<u>RAP Architecture</u>. RAP was developed at the University of Toronto in the mid 1970s (Ozkarahan, 1975; Schuster, 1976). The overall RAP architecture is illustrated in Figure 18. The host is responsible for compiling the high- level user queries into RAP commands, scheduling the operations for RAP, transmitting the RAP instructions to RAP, handling all database integrity and security, and maintaining all the relation and domain tables. RAP consists of a controller, a set function unit (SFU), and a number of cell processors, each connected to its two adjacent neighbors.

The design is composed of a controller, an arithmetic set function unit, and a parallel organization of cell processors (Figure 19). A cell processor consists of a memory component and a logic component. The memory unit is one track of a rotating device such as a disk, drum, circular shift register, etc. The logic component is a microprocessor which acts as a "search machine" on data, directs manipulation, and performs limited numeric computations required by database processing. The set function unit is used to combine cell processor results to obtain a value computed over the total memory contents. The controller is responsible for overall coordination and sends control sequences to the cell processors, controls the set function unit, and executes decision commands and other RAP primitives that can be accomplished directly in

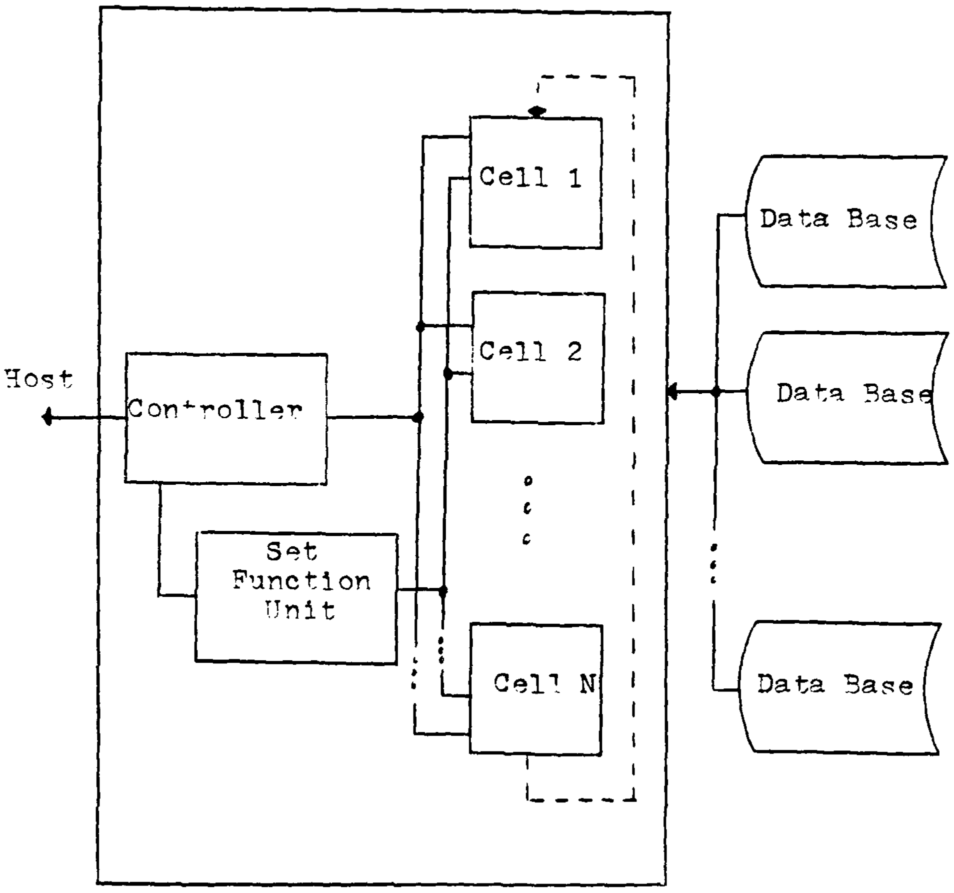


Figure 18 The RAP System Architecture (Bray, 1979:126)

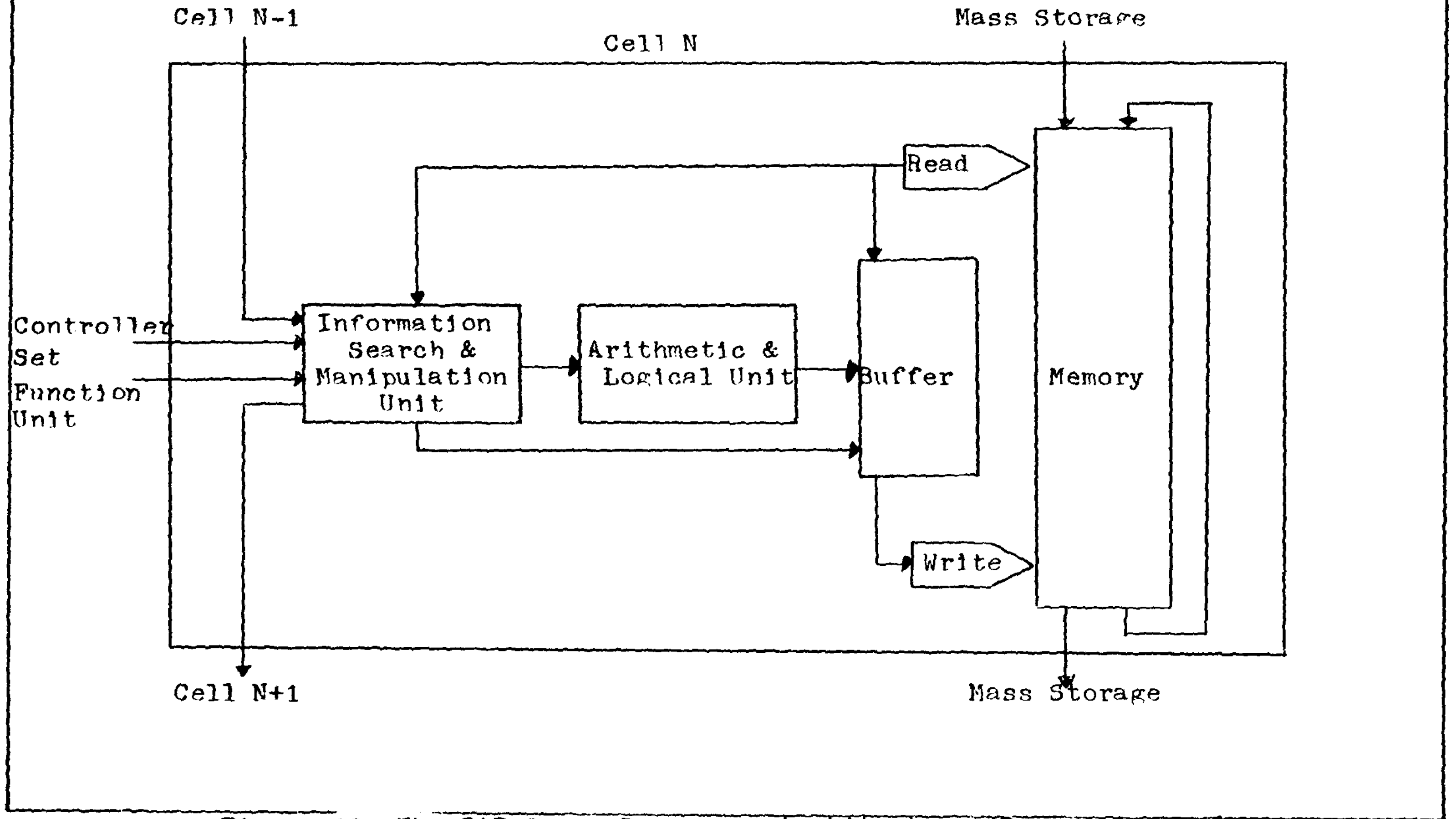


Figure 19 The RAP Query Processor Architecture (Bray, 1979:126)

itself.

RAP is based on the relational data model, with the data stored in normalized relations. Only one type of relation can be stored on a single track. If there are too many tuples for a single track, then any number of additional tracks can be used. Neither contiguous tracks nor ordered tuples are required. If the RAP database is small enough, it can be stored completely within the cell processor's memories and operate as a direct- search system because staging is not required. The more general case is that the database is too large and must be stored on a conventional moving-head disk.

Intermediate storage within each RAP cell processor contains the equivalent of one disk track of rotating memory, for example, disk, CCD, or MBM. Like CASSM, the data, rotating through the intermediate storage, are read into the cell processor's buffer, processed, and written back to intermediate storage. The tuple size is limited to the buffer in the cell, for example, RAP's buffer size was 1024 bits.

<u>DIRECT Architecture</u>. DIRECT was developed at the University of Wisconsin (DeWitt, 1979) and is currently in the stage of implementation refinement (Boral, 1980; Boral, 1981). The overall DIRECT architecture is illustrated in Figure 20. Where RAP is a SIMD approach only able to process one instruction at a time, DIRECT is an MIMD approach and so is capable of simultaneous execution of

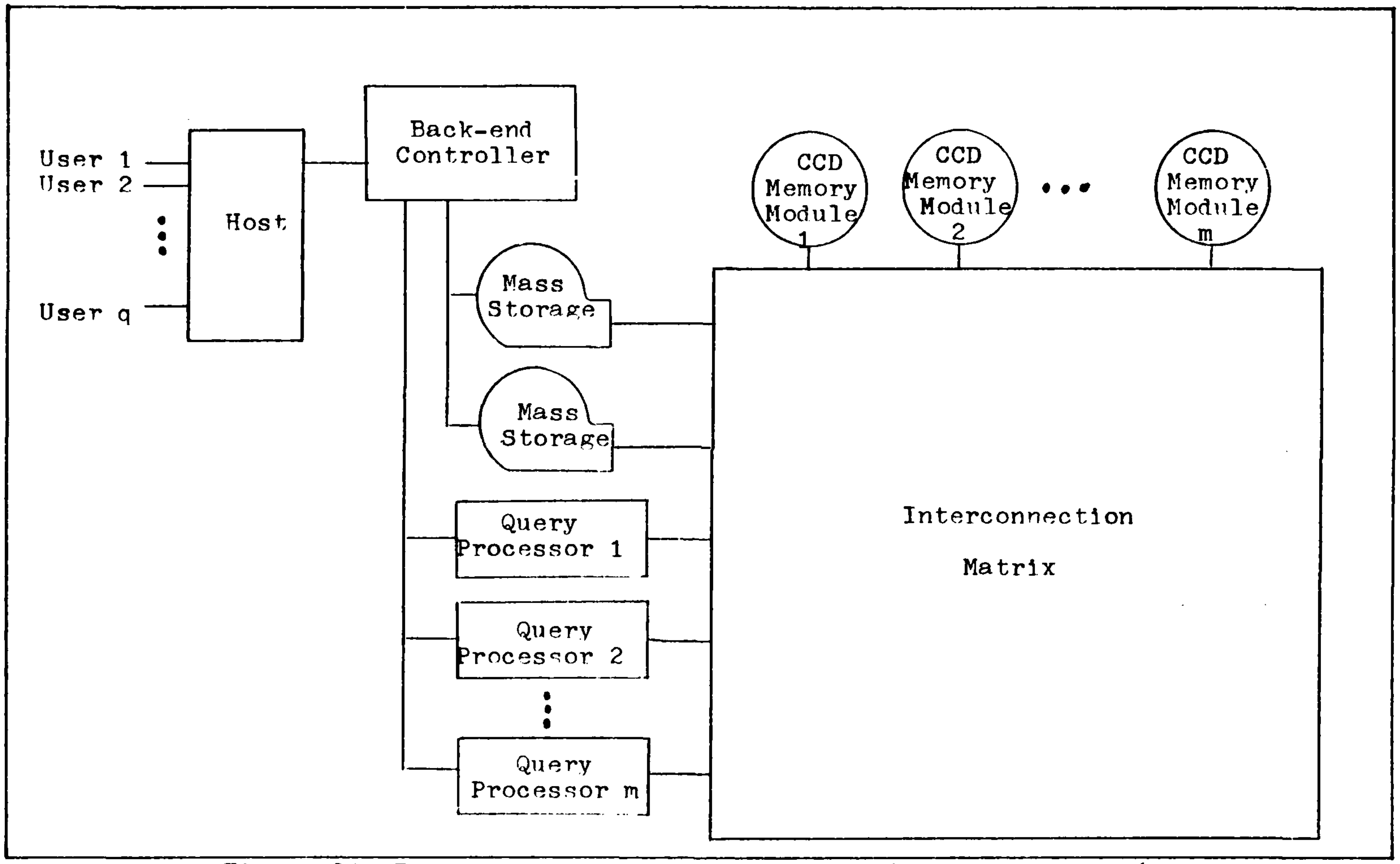


Figure 20 The Direct System Architecture (DeWitt, 1979:396)

query instructions from different users. DIRECT is very similar in operation to RAP with the exception that the intermediate storage units are not specifically assigned to a query processor. Through the use of an interconnection matrix, any intermediate storage unit is accessible by any query processor. The controller makes dynamic determination of the number of processors assigned to each query to be processed.

Using DMA data transfer, the controller stages tracks from main storage to the intermediate storage units and informs the query processors of which intermediate storage unit to access in order to process that information. As with RAP, only one relation is stored per track. The intermediate storage units were constructed using CCD chips. The relations are not assigned to a query processor, but instead, a data flow technique is used in which the query steps are assigned and processed by the whole set of query processors.

<u>Multiple Processor Combined Search</u>. The multiple processor combined search (MPCS) approach combines several of the best features of both the direct- and indirect-search approaches. This approach handles the complexity/performance tradeoff issue by directly searching the data with track processors for simple queries and staging the data in buffers for more complex evaluations.

<u>DBC Architecture</u>. DBC (Data Base Computer) is an example of the MPCS approach (Banerjee 1979). It is under

development at the Ohio State University. It has three major objectives: 1) to support very large databases of 10 to 100 billion bytes, 2) to support multiple data models, including hierarchical, network, and relational, and 3) to use current technology and not rely on significant technological breakthroughs.

Modified moving-head disk technology is used in order to support the very large on-line database storage. DBC employs two forms of parallelism. An entire cylinder's set of tracks can be processed in parallel by a processor assigned to each track within the disk pack. DBC uses a pipeline architecture which provides a separate unit to process each step of an instruction. The basic architecture is conceptionalized in Figure 21. The DBC makes use of two loops of processors and memories in executing the commands. The data loop, which consists of the database command and control processor (DBCCP), mass memory (MM), and security filter processor (SFP), is used for storing and accessing the database, for post processing of retrieved records, and for enforcing record field level security. The structure loop, which consists of the database command and control processor (DBCCP), keyword transformation unit (KXU), structure memory (SM), structure memory information processor (SMIP), and index translation unit (IXU), is used for limiting the mass memory search space (through the determination of cylinder numbers), for determining the authorized records for

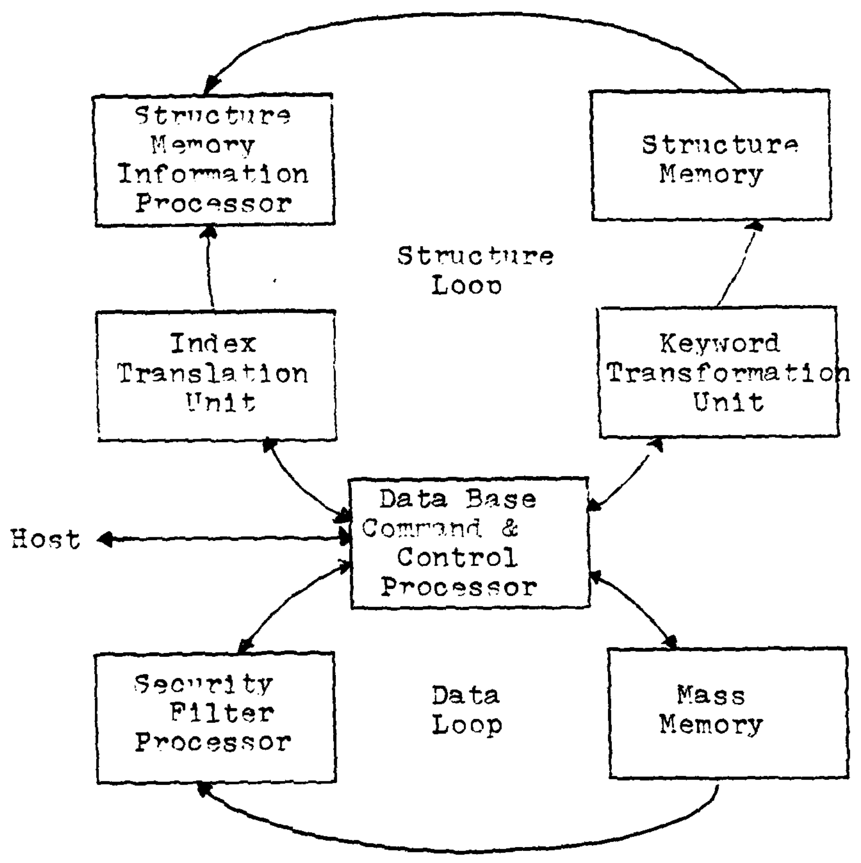

Structure Memory Information Processor
Structure Memory
Structure Loop
Index Translation Unit
Keyword Transformation Unit
Host
Data Base Command & Control Processor
Security Filter Processor
Data Loop
Mass Memory

Figure 21 The DBC System Architecture (Banerjee, 1979:415)

accesses, and for clustering records received for insertion into the database. The system was designed completely around the security design to enable full security capability for this system. This system can process multiple queries at a time.

Comparisons. Note that each of the categories' architectures presented have been built upon the architectural design of the previously presented category. It so happens that the ranking of these categories correspond to the sequence of events in the research and development in this field of computer science. Each of these research efforts have produced respectable results as well as insights and questions bringing on further investigation and development of yet more sophisticated architectural designs. As comparisons are made and discussed in these next paragraphs, the motivations behind these successive developments will come to light.

In terms of response time, the MPDS architecture offers the best performance by allowing the entire database to be searched in one mass storage revolution. As in CASSM's case, since the data definition is carried with the data on mass storage, no indexes of any kind are required - the data may reside anywhere on disk - all within reach of the query processors within one mass storage revolution. The drawbacks to this system are 1) such a system is currently only effective for small databases, because of its higher storage costs, and 2) due to the constant rewrite of the entire database, recovery becomes a nightmare. Such an architecture is an excellent advancement over previous efforts, in terms of using a direct access mechanism along with a set of processors, but there was plenty of room for improvement.

The next logical step was to use a data-staging technique and stage organized blocks of data from mass storage into an intermediate storage. This approach solves the growth/cost penalty problem substantially and now also enables conventional recovery techniques to be used since instantaneous rewrite is no longer being accomplished. Thus an [illegible] architecture is created. But there is concern regarding the reduction in response time due to the additional process of staging blocks of data to and from mass storage. Whether or not a reduction does occur boils down to whether a SIMD or MIMD architecture is used. RAP, though dealing with an organized database (relations stored on separate mass storage tracks), only processes one instruction at a time. Relations whose total number of blocks is less than the number of query processors available, and relations whose total number of blocks divided by the number of query processors available yeild a large remainder, will render the remaining available query processors useless for all or part of the duration of that query processor instruction. In this case, response time would be reduced - all other factors being equal. By adding more intelligence to the controller processor, and making use of an already existing relation block index structure, query processors found to be idle could begin processing another query instruction of the same or a different query, and thus boost the performance of the system, as in DIRECT's case. Thus another step forward has again taken place, in this case, from a MPIS (SIMD) architecture to a MPIS (MIMD) architecture. Again, each development has produced excellent results over previous efforts and agreeably, without these valuable stepping stones, progress would not have come this far at this time.

There is plenty of agreement that excellent benefits exist in both the MPDS and MPIS architectures, so why not put to use the best of both architectures. Instead of just transferring the blocks of data from mass storage to the intermediate storage units, why not perform some level of direct search processing on that data as it streams off the track over to intermediate storage, and then proceed with the more complex query processing on the staged blocks of data. Thus the MPCS architecture, specifically DBC's case, is created. Where one goes from here in terms of new architectural approaches is yet to be conceived. But in the mean time, by concentrating on improving the building blocks themselves that will make up these present approaches, much can still be accomplished. By incorporating more sophisticated CAM, CCD, MBM, etc. into these architectures, both in terms of quality and quantity

(VLSI), query processing response time for laıge database systems using the relational model, will easily be achieved.

Summary

The first of four problems identified in the purpose of this thesis effort was to determine the feasibility of applying multiple processor techniques to the implementation of a relational DBMS within a micro-/mini-computer system environment. Based on the information brought to light within this background chapter, the feasibility of such an architecture is now a certainty. The remaining chapters in this thesis address the last three problems identified in the purpose of this research effort.

III. System Development

Introduction

This chapter covers the initial system development for an AFIT Backend Relational DBM computer system. In the first section, the longterm requirements and goals are identified, first in general terms and then in more specific terms. Justification of these requirements and goals is given. In the next section, at the overview level, the system design is presented. The system functions and interrelationships, the system hardware architecture, and the system support software are defined. In the next section, a system development plan is presented. Based on the system design, a set of subsystem development stages are identified. The resources and constraints for each stage are also identified so as to specifically define what functions will be implemented or must be simulated for each stage of the system development. A summary concludes this chapter.

Longterm Requirements and Goals

The longterm requirements and goals for the development of an AFIT Backend Multiple-Processor Relational DBM computer system have been determined to consist of the following items.

(1) Permit a multiple number of users to share the database as a common resource (i.e.: a network member).

(2) Use a MIMD approach to permit a high level of

parallel processing of users' queries.

(3) Design a multiple processor system which does not waste a large percentage of its processing potential by permitting processors to be idle, particularly in a multi-user environment.

(4) Considerably improve on the system response time, system throughput, and cost of database storage in comparison with existing software laden DBMS's run on general purpose computers.

(5) Provide complete modularity in both the software and hardware design of the system in order to enable it to (a) be easily modified throughout its initial development and (b) be used as an easily modifiable AFIT research test bed for investigating and implementing alternative state-of-the-art architectures.

(6) To be used in both an AFIT pedagogical environment and an actual AFIT supportive information processing environment.

Let it be emphasized that the central goal is to develop a DBMS that can provide a very respectable online response time to a very large variety of queries generated by a multiple number of users against a large database. From the background material presented in Chapter II, it should be clearly evident that to accomplish this central goal, the use of the relational data model, a backend computer system composed of a set of processors, and some type of intermediate associative memory storage is, to date, the only real route to take in order to achieve the speed and size qualities desired.

Based on an analysis made of all the key features of backend computer systems, the relational data model, state-of- the-art memory technology, and the different relational DBMSs implemented on special purpose hardware, a set of functional and architectural features were selected for their inclusion in the development of a backend DBMS in this thesis. Specifically, the design of the backend multiple- processor relational DBM computer system, as currently envisioned, will include the following features.

(1) A complete set of relational algebra based operations will be provided. Having a complete set means that any derivable relation can be derived from the database. Therefore, if the information wanted by the user is in the database, it can be retrieved.

(2) A set of N query processors in the backend computer architecture will be provided to support the parallel processing of a single relational query. Three levels of parallelism are capable of being performed on relational algebra queries - independent parallelism, pipelining, and node splitting. Exploitation of these levels using multiple processors will make highly parallel processing and short response times possible.

(3) One or more mass storage devices to contain the entire relational database will be used. This will require using moving- head disk devices.

(4) A multiple access path to the database will be provided so as to fully support the parallel processing of a single relational query. Having simultaneous access to the database by the N query processors will provide a complete parallel processing capability (data access and data processing).

(5) A set of M (where M >> N) intermediate memory modules will be provided to enable the physical concurrent access of the relational database. By staging relations to separately accessable memory modules, the query processors can then concurrently access and process the data.

(6) Enable the size of a relation to surpass the size of the memory module device. By dividing the relation into logical pages and staging these pages to the intermediate memory modules, the size of a relation need not be restricted.

(7) A dynamic determination of relation page assignments to query processors will be provided. An entire relation does not have to be processed by a single query processor for a given relational query operation, but can be paged out to N processors and processed in parallel.

(8) A dynamic determination of the number of processors assigned to process a relational query will be provided.

(9) A set of N query processors, each executing the same query step, will be permitted to simultaneously search different pages of a common relation.

(10) A set of N query processors, each executing different queries, will be permitted to simultaneously search the same page of a relation.

These are the longterm requirements and goals, both broad and specific, for the development of the backend relational DBM computer system. While the capabilities of this system are quite clearly defined, the method of implementation of certain parts of the architecture leaves room for several alternative approaches. They are as follows.

(1) Type of intermediate storage modules to use.

(2) Hardware configuration and method of data transmission in the staging of relation pages from the mass memory device to the intermediate memory modules (IMM).

(3) Hardware configuration and method of data transmission in the accessing of relation pages from the IMMs by the query processors.

(4) Hardware configuration and method of data transmission between the query processors and the backend controller processor, and between the host computer and the backend controller processor.

Random access memory, associative memory, charge-coupled memory, magnetic bubble memory, hard disk memory, or floppy disk memory could be used for the intermediate memory modules. To handle data transfer between the main storage and the IMMs, and the IMMs and the query processors, either multiport memory modules or a cross-point matrix switch could be used. Programmed I/O, direct memory access (DMA), or memory module continuous data broadcasting could then be used for actual data transfer.

For the initial design, multiport memory modules will be used. Figure 22 shows the conceptional design approach. This configuration will lend itself to easy modification in many areas of the hardware architecture. These details will be specifically discussed in a later section of this chapter.

System Design

General Description.

This Backend Multiple-Processor Relational DBM computer system will consist of five main components: a host computer, a backend controller processor, a set of query processors, a set of dual-port intermediate memory modules (IMM), and one or more mass storage devices. A diagram of these components and their interconnections is shown in Figure 23.

The host computer will handle all communications with the users of the DBMS. Users will log onto a modified version of Roth's Relational DBMS (Roth, 1979), and proceed in the normal manner. When a user wishes to execute a query, Roth's DBMS will format the already optimized user query into a sequence of relational algebra operation steps, called a "query packet", and then send it to the backend controller processor over a communication link. Query responses are received by the host computer from the

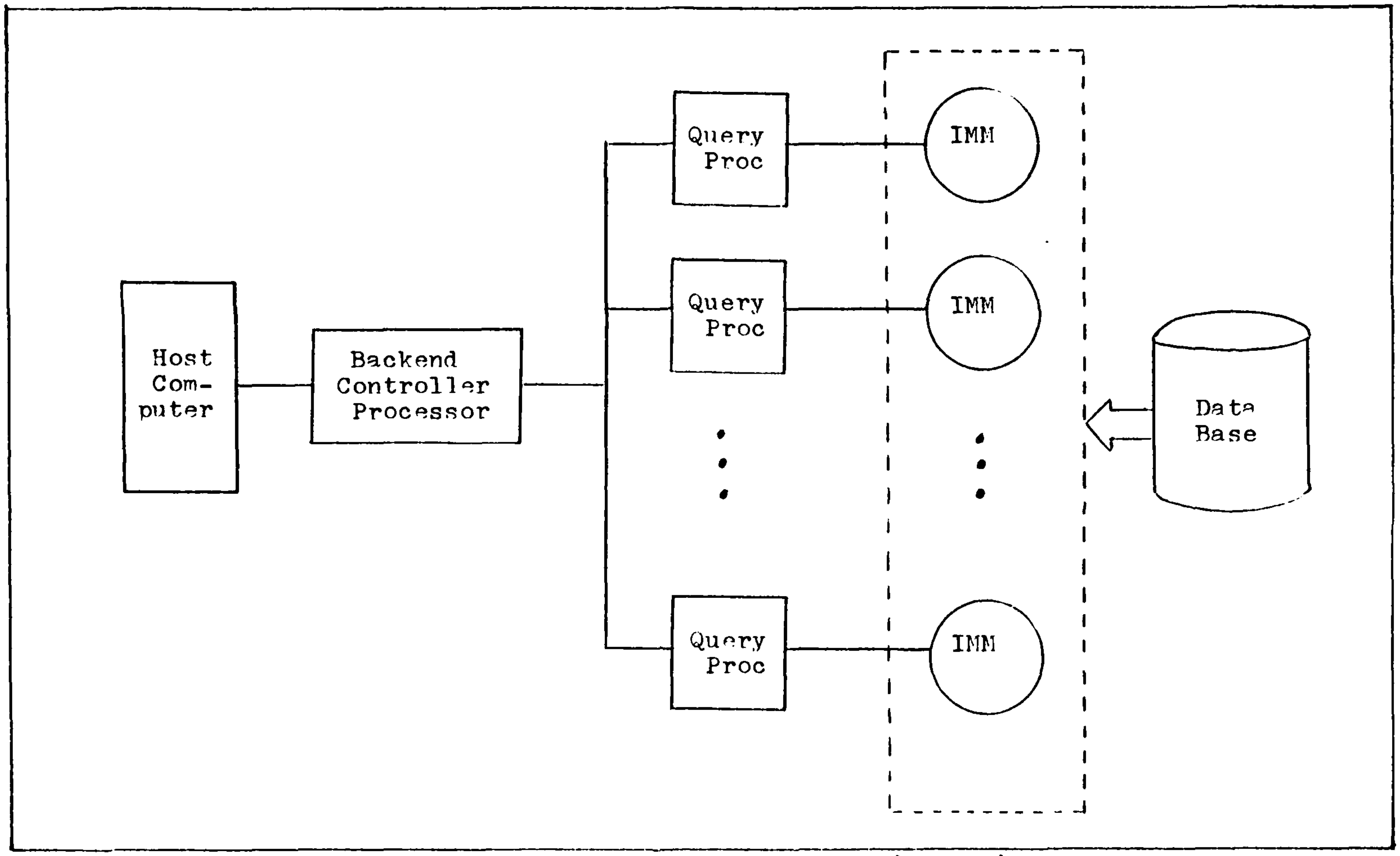


Figure 22 The Conceptional Design Approach

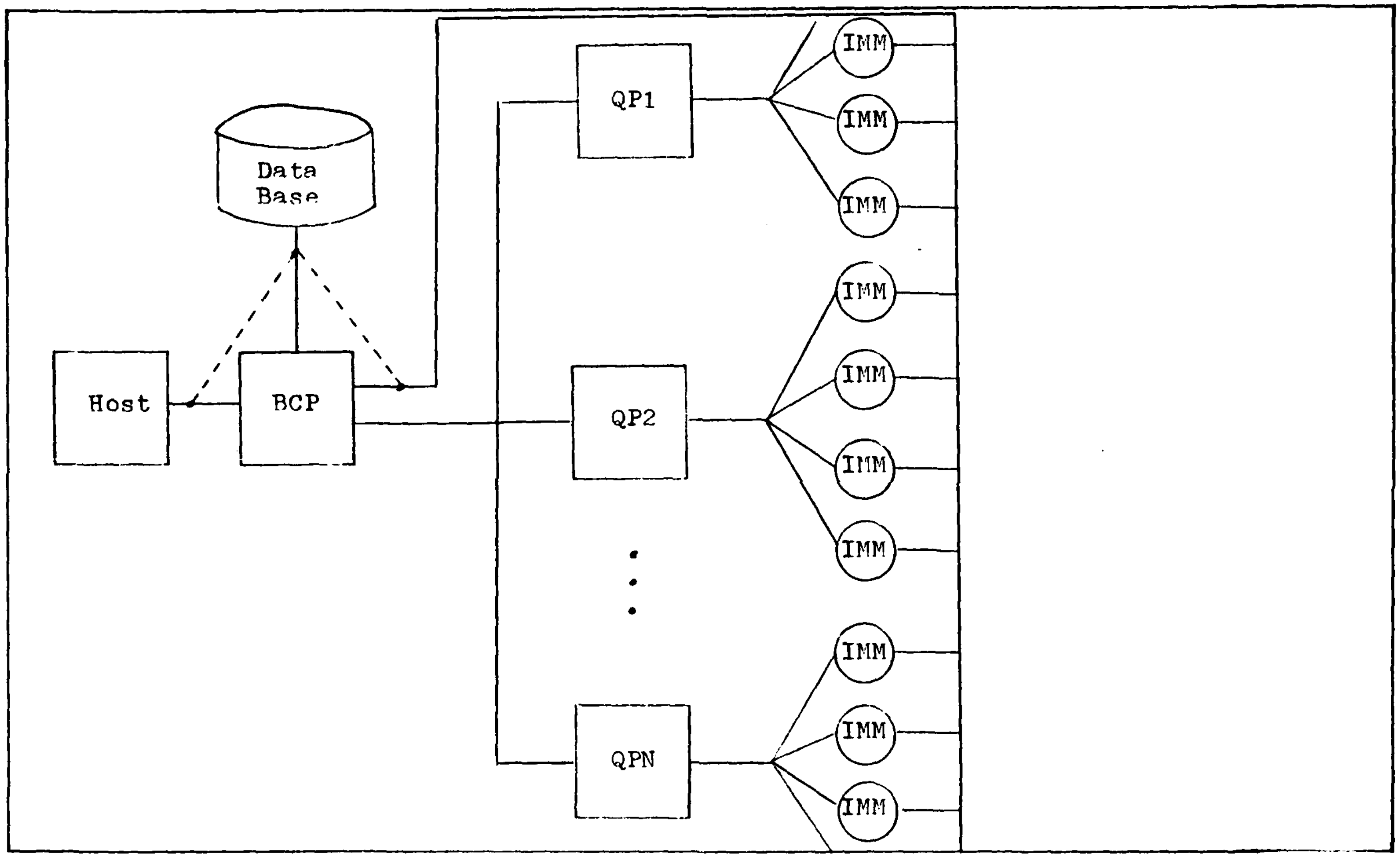


Figure 23 The Physical Design Approach

backend controller processor via DMA data transfer from the BCP's disk storage. The query responses are then subsequently passed back to the awaiting user.

The backend controller processor is responsible for interacting with the host computer, controlling the query processors, and handling the transfer of data between the mass storage and the IMM devices. When the backend controller processor receives a query packet from the host computer, it will determine the number of query processors that should optimally be assigned to the query packet. Before distributing the query packet to the selected query processors, the backend controller processor will page, via DMA data transfer, portions of the required relations into the proper IMMs. During execution of the query packet steps by the query processors, the backend controller processor will continue to supply needed relation pages for processing in response to query processors's requests.

The function of each query processor is to execute the steps of the query packet assigned by the backend controller processor. The query packet is received from the backend controller processor over a communication link to the query processor(s). The query processors then receive from the backend controller processor the address of which IMM to access to process the query step (multiple IMMs can either be permanently dedicated to each query processor or dynamically assigned by the backend controller processor). Intermediate query results (temporary relations) are written by the query processor out to another IMM, also specified by the backend controller processor.

To enable the support of both intra- and inter-query concurrency of processing, the approach taken was to divide each relation into fixed size pages and also divide the memory into separately addressable memory modules having the same size as a relation page. These relation pages are then staged from the mass memory to the IMM modules and also written back if updates were performed. Each query processor will search an assigned page (staged in an IMM) of a relation referenced in the query packet step. When a query processor finishes searching one page of a relation, it will then make a request to the backend controller processor for the address of an IMM containing the next page it should search. Once the address has been received, the query processor must be able to rapidly switch to that IMM device.

A relatively small page/IMM size shall be chosen. By choosing a small page size, and having a large number of IMMs, one will have a higher potential for concurrency of processing. If the page size were too large then many of the relations might fit on just a couple of pages. This would limit the potential concurrency to just inter-query concurrency instead of a mix of intra- and inter-query concurrency. Another important reason for choosing a small page size is to minimize the amount of internal fragmentation which occurs when a relation does not fit all of the pages it occupies. This could result when N query processors, each processing the same query operation, generate a partially filled page for the resulting temporary relation. On the other hand, one must realize that by having relatively small page sizes, many more pages will have to be staged, resulting in a higher level of page staging overhead for the backend controller processor. Both the specific configuration of the system's architecture (number of query processors) and the characteristics of the database (size of relations) will be determining factors in choosing a page size.

It is recommended that eight query processors and 48 IMMs be used in this implementation. Since both intra- and inter-query processing is to be supported, anything less than eight query processors would be insufficient in supporting the concurrency concepts as described in the Relational Data Model section of Chapter II. The requirement of the number of IMMs is directly based upon the number of query processors to be used. Forty- eight IMMs are required for the following reasons. Eight IMMs would always be under current access (one per query processor); eight more modules would contain the next page ready to be processed (anticipatory paging); 16 more modules would be required in readiness to accept temporary relation pages, and finally, 16 more modules (8 for current pages and 8 more for next pages) would be required to contain a page of a relation which is to be joined with the current page being accessed.

The bandwidth of the intercommunication scheme must be very high. Dual-port intermediate memory modules (each connected to a single query processor and to the backend controller processor) would be suitable for a medium-scale implementation, but for a very large configuration a cross-point switch (each query processor is connected to all the intermediate memory modules) looks to be the only feasible interconnection scheme. Traditionally, the use of cross-point switches has been limited because of their high cost and complexity due to the following requirements.

(1) High bandwidth between processors and memories for addresses and data.

(2) Extremely fast switches to minimize the delay time introduced by the switch in each memory access.

(3) Contention detection and resolution hardware to handle simultaneous access of two or more processors to the same memory bank.

However, recent research in the design of cross-point switches specially built for use in backend multiple-processor database computer systems is occurring, in which these traditional limitations are being overcome (Dewitt, 1979: 398). These new designs are now making such an approach a very viable solution for large scale implementations; i.e. 64 processors x 384 memory modules.

System Functions. The following three sections identify the backend multiple-processor relational DBM computer system's functions as they relate to the three major system components.

(1) Host Processor

(a) Convert the optimized query into a sequence of relational algebra operation steps.

(b) Construct a query packet message

(c) Place the query packet message into a query packet queue within priority of query.

(d) Recieve query response messages from the backend controller processor.

(e) Format query responses for shipment to the user.

(f) Ship the query response to the awaiting user.

(2) Backend Controller Processor

(a) Access the query packet queue and move the query packet message into a query packet execute table.

(b) Perform a security access rights check on the user request. If access is unauthorized, reject the user query.

(c) Perform a validation check on the query. If the query is incorrectly formatted, reject the user query.

(d) Examine a query packet and estimate the optimal number of processors to allocate for processing the packet; update the query packet execute table.

(e) Perform staging of relation pages from mass memory into the intermediate memory modules.

(f) Assign a query packet to query processors.

(g) Transmit query operations and relation's tuple format to the query processor.

(h) Perform anticipatory staging of relation pages from mass memory to the intermediate memory modules.

(i) Control the locking and unlocking of relations to enable the updating of a relation.

(j) Respond to a "next-page" request from a query processor.

(k) Respond to a "get-page m" request from a query processor.

(l) Perform the rewrite of relation pages from the intermediate memory modules to mass memory.

(m) Create new relations as a result of temporary or permanent relations generated by the query processors.

(n) Delete relations which are no longer needed.

(o) Print relations, i.e. access an entire relation for its subsequent transmission back to a host's user.

(p) Perform a compaction/reorganization on a relation to minimize page seqmentation.

(3) Query Processors

(a) Recieve a query packet from the backend controller processor.

(b) Retrieve the "next-page" address from the backend controller processor.

(c) Request the "next-page" address from the backend controller processor.

(d) Request a "get-page m" address from the backend controller processor.

(e) Switch to address of intermediate memory module

(f) Access the contents of the intermediate memory module.

(g) Write temporary relation tuples into the intermediate memory module.

(h) Perform the following relational algebra operations.

    (1) Select
    (2) Project
    (3) Join
    (4) Union
    (5) Modify
    (6) Add
    (7) Delete
    (8) Maximum Value
    (9) Minimum Value
    (10) Count Value
    (11) Average Value

System Hardware Architecture. The architecture of this backend DBM computer system is categorized in Bray's classification scheme under the category entitled "Multiple Processor - Indirect Search" and is also categorized in Flynn's classification scheme under the category entitled "Multiple Instruction Stream - Multiple Data Stream (MIMD)." An architecture using a set of processors and a data staging methodology enables the system to be functionally designed to accomplish concurrent processing of individual query operations from one or more user queries, thus providing a greatly reduced and now respectable response time to user queries made against a large online data base.

The host computer, containing the user's application software and, if a network member, containing the network interface software, will be responsible for passing and receiving queries to and from the backend controller processor. To handle such a workload and not be a bottleneck in communicating with the backend controller processor, the host computer needs to support a multiprogramming environment and be directly coupled to the backend controller processor by sharing memory and using direct memory access (DMA) data transfer. By using DMA having a high bandwidth and using interrupt circuits, a much greater speed of I/O operations between these computers can be achieved while at the same time freeing up both CPUs to do more important processing instead of having to be tied up with executing routines that determine I/O device status.

The backend controller processor will contain the major portion of the software written for the backend system. The relationship between the backend controller processor and the query processors will be a Master - Slave relationship: the backend controller processor is the master and each query processor is the slave. Since the master has eight slave processors to keep busy, instructions and data must be supplied to the processors very quickly in order to uphold the high degree of parallelism desired. In order to quickly satisfy the query processor requests for relation pages, the backend controller processor will (a) be connected to each of the query processors in an indirectly coupled fashion by using channel-to-channel adapters having a high bandwidth, and (b) be connected to each of the intermediate memory modules in a directly coupled fashion using DMA transfer between the mass storage device(es) and the IMMs. Since messages between the backend controller processor and the query processors, though frequent in nature, will be quite small in size, programmed I/O (from BCP to QP) and interrupt I/O (QP to BCP) using a channel connection (one for each QP) will be quite sufficient. On the other hand, relation pages being staged to and from mass storage and the IMMs will be both frequent and relatively large in size, thus requiring the use of DMA transferring. A number of the

AD-A115 558 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
DESIGN AND IMPLEMENTATION OF A BACKEND MULTIPLE-PROCESSOR RELAT--ETC(U)
DEC 81 R W FONDEN
UNCLASSIFIED AFIT/GCS/EE/81D-6 NL

2 of 3
AD
A 115558

primary software functions will be interrupt driven by the query processors.

The query processors will be responsible for the processing of existing relations and the generation of new relation pages (for both temporary and final relations). A microprocessor will be used for each of the query processors, each containing enough local memory to contain its software plus working storage.

Since the query processors and the backend controller processor both require access to the IMMs, a dual-port IMM device will be used. Each query processor will have access to six of these dual-port IMMs. Since the query processors will function as slave processors to the backend controller processor, the case will never occur where a query processor and the backend controller processor both concurrently try to address the same IMM. The query processor will always be told in advance which IMM to access.

The intermediate memory modules may be built using content-addressable memory (CAM), charge-coupled memory (CCD), magnetic-bubble memory (MBM), random-access memory (RAM), or disk memory. The requirement of having dual-port access to the IMM is the only requirement which must be met.

System Support Software. The system software for these three computers should be written using both a high-level programming language and an assembly language. A high-level language such as PASCAL should be used for the major part of the software because it is a block structured language which is very compatible with top-down pseudo-code written design documentation. An assembly language will be required for writing the low level program/hardware interface routine modules.

UCSD PASCAL, which consists of an operating system, the PASCAL language, a screen-oriented editor, a linker, and a debugger, will be used for this development effort (UCSD, 1979). This is a very capable development system and is used by a wide variety of micro- and mini- computers.

Selection of an assembly language will solely depend on the choice of computers used for each of the processors to be implemented in this development effort.

System Development Plan

General Description. Due to the complexity and size of this computer system, a plan of attack has been formulated as to specifically what the sequence of development must be to effectively design, build, and test this system. This sequence has been carefully grouped into development stages.

It is imperative that the documentation for each stage of this development effort be accomplished in a thorough manner, and that a top-down structured design and coding approach be taken. If this is not rigorously followed, development effort in succeeding stages of the plan may suffer.

Development Stages. The stages of development for the construction of this computer system are as follow.

Stage (1) Design and implement the functions of the backend controller processor.

- (a) design and write the software for the BCP
- (b) simulate a DMA interface to/from the host computer by using a parallel link
- (c) simulate the host computer and software
- (d) simulate the QP and its software
- (e) simulate the IMM and its software
- (f) simulate a DMA interface from the mass storage device to the IMMs using a set of serial links
- (g) simulate a parallel interface to the query processors using a set of serial links

Stage (2) Design and implement the functions of the query processor.

- (a) design and write the software for the QPs
- (b) talk to the backend controller processor using the current interface
- (c) simulate access to the IMMs using serial links

Stage (3) Design and implement the modifications to Roth's Relational DBMS.

- (a) design and write the software
- (b) interface with the backend controller processor

(c) interface with the AFIT local network

Stage (4) Design and implement the following.

(a) the DMA link between the host and backend controller processor

(b) the set of parallel links between the backend controller processor and the query processors

(c) the dual-port IMMs and the DMA and parallel communication links to the processors

Stage (5) Develop and test alternative parallel processing optimization algorithms.

Stage (6) Develop and implement the IMM devices using one of the following technologies.

(a) MBMs

(b) CCDs

(c) CAMs

(d) head-per-track disks

Stage (7) Develop and test a cross-point matrix device.

Stage (8) Simulate alternative system configurations in search of an optimized system configuration i.e. numbers and ratios of query processors and IMMs; sizes of IMMs; connection scheme between query processors and IMMs; mix of simple and complex relational queries.

Note that this sequence of development stages does not have to be accomplished in a totally serial fashion. Stages one ,two and three may be accomplished in parallel, and stages five and eight, six and eight, or seven and eight may be accomplished in parallel. Stage seven would be dependent upon whether or not stage six was accomplished.

Resource Requirements. The resource requirements must be outlined as to what equipment will be required for each stage of the development effort as well as the time frame, otherwise the work to be done will be not only difficult to accomplish, but would eventually come to a halt. The resources required for each stage of the development effort are as follows.

(1) Stage 1

  (a) The microcomputer to be used as the backend controller processor.

  (b) a micro-computer to simulate the host

  (c) a micro-computer to simulate a query processor(s)

  (d) a serial interface to the query processor(s) and a parallel interface to the host computer

  (e) a micro-computer to simulate an IMM(s)

  (f) a serial interface to the "IMMs" from the BCP and the QPs

(2) Stage 2

  (a) The microcomputer to be used as the query processor.

  (b) items 1a, 1b, 1d, 1e, 1f.

(3) Stage 3

  (a) The computer to be used as the host system

(b) the local network interface communication link.

(c) items 1a, 1d.

(4) Stage 4

(a) items 1a, 2a, 3a.

(b) DMA boards and links for the host/backend interface and backend/IMM interfaces

(c) Parallel port boards and link for the backend/query processor interfaces and the query processors/IMM interfaces.

(d) dual-port IMM devices

(5) Stage 5

(a) A completely functioning system composed of eight query processors and 32 IMMs.

(6) Stage 6

(a) The complete system

(b) the dual-port IMMs chosen to implement

(7) Stage 7

(a) The complete system

(b) the parts to construct a cross-point matrix

(8) Stage 8

(a) access to QGERT, SLAM, or other capable simulation language.

(b) access to a computer system which supports the chosen simulation language.

Summary

The second and third problems identified in the purpose of this effort were to 1) determine the system requirements, both the longterm and shortterm requirements and goals, and 2) develop a structured, modular system design (software and hardware using current state-of-the-art technology) for implementing the required system over the longterm, identifying experimental tradeoffs. Both of these problems have now been fully addressed. The system requirements were thoroughly defined, and then, based on these requirements, a system design was developed. A general description was given, its system functions defined, its hardware configuration defined (including alternative tradeoffs) and the development support software specified.

Toward the end of this development process, it became fully evident that the time necessary to accomplish the detailed design, implementation and testing of this system design would indeed span several thesis efforts worth of work. Thus a system development plan was created; a plan of attack specifying an organized sequence of development stages (thesis efforts) for designing, building and implementing this system; each stage building upon the accomplishments of the previous stage.

The remaining chapters of this thesis address the fourth and final problem defined in this thesis: the detailed design and implementation of the backend controller subsystem (the heart of the backend data base computer system), thus establishing the initial stage and cornerstone of this system.

IV. Subsystem Design

Introduction

This chapter describes in more detail the design of the software developed for the backend controller processor subsystem of the backend multiple-processor relational data base computer system.

For ease of discussion and reader comprehension, the relational query operation formats will first be described, next the BCP subsystem data tables shall be identified and discussed; then the BCP subsystem functions and logic flow will be presented; and finally the message formats for inter-processor communication will be described.

Query Operation Formats

As identified earlier, the relational operations to be supported in this implementation are: select, project, join, union, modify, add, delete, maximum, minimum, count, and average. The relational algebra (RA) data base sublanguage was chosen to be used because in being an algorithmic approach rather than a mathematical non-procedural approach, (as in relational calculus), the RA data base sublanguage is both easier to learn and easier to implement and use.

The operands for each of these RA operations consist of variable byte length fields, in order to avoid wasting space. Delimiters are used to indicate the end of each operand field. Appendix B contains the detailed formats for each of the RA operations when transmitted from the host to the BCP.

Subsystem Data Tables

The subsystem data tables are the foundation upon which the subsystem functions are constructed and are able to accomplish their tasks. Since this is a relational data base system, care has been given to minimize the number, size, and use of system tables in support of the data base processing operation.

Twelve BCP subsystem data tables, maintained in main memory in linked list form (built via dynamic memory allocation), are used to accomplish the functions within the BCP subsystem. These data tables consist of the following.

1) domain name table
2) attribute name table
3) relation table
4) relation's attribute table
5) relation's page table
6) query processor table
7) query processor's IMM table
8) query packet wait queue
9) query packet operation list
10) query packet execute list
11) query operation state table
12) transmit queue

The specific formats for each of the above subsystem data tables are described in Appendix C.

Domain Name Table. The domain name table consists of a complete dictionary list of all domains contained in the entire relational data base. Attributes contained in relation tuples are defined by specifying a "domain" of values which they may take on. Each entry consists of a domain name and a linked list of attribute name entries currently defined against this domain set. This table is used to support the validation editing of all attributes referenced in query operation "join" steps of a query request submitted to the BCP.

Attribute Name Table. The attribute name table consists of a complete dictionary list of all attributes contained in the entire relational data base. Each entry consists of an attribute name, the number of bytes long the attribute value may consist of, and the data type the attribute represents. This table is used to support the validation editing of all attributes referenced in each query operation step of a query request submitted to the BCP.

Relation Table. The relation table consists of a list of all relations making up the relational data base. Each relation entry consists of a relation name, a relation code number, a pointer to its relation attribute table, a pointer to its relation page table, a lock indicator, a lock owner identifier, the byte size of its tuples, and the number of attributes which make up each tuple. The relation table is used for validation editing of query requests and for processing the query processor requests from the QPs which process the query operation steps. It is through this table that access is made to the relation's attribute table and page table, to accomplish preparation of query requests for submittal to the QPs and later to accomplish the location, creation, staging and rewriting of relation pages.

Relation Attribute Table. A relation attribute table exists for each of the relations within the relational database. Each relation attribute table consists of a list of all the attributes which make up each tuple of the relation. Each relation attribute table is accessed through a pointer which exists within each entry in the relation table. Each entry of the relation attribute table consists of an attribute name, its byte offset within the tuple, the byte length of the attribute, the data type that the attribute represents, and a key member indicator. These tables are used for query validation editing and query format preparation for submittal to the query processors.

Relation Page Table. A relation page table exists for each of the relations within the relational data base. Each relation page table consists of a list of the page addresses for all the pages which make up the relation's tuples stored in mass storage. Each relation page table is accessed through a pointer which exists within each relation entry in the relation table. Each relation page table entry consists of a page number and an address of where to access the page with mass storage. For this initial implementation, each page of a relation will consist of a PASCAL file residing on mass storage. These relation page tables are used to support all QP requests to locate, create, stage, and update relation pages being concurrently processed by the QPs in processing the query request(s).

Query Processor Table. The query processor table consists of a list of all the query processors assigned to the BCP to be used to process the query packet's operations. This table is used to maintain the state of each query processor, either in use, idle, or not on-line. It is through this table that the BCP can assign idle QPs to process a query request. Each query processor entry consists of a QP number identifier, a state indicator, and a pointer to its IMM table.

Intermediate Memory Module Table. An intermediate memory module table exists for each of the query processors assigned to the BCP. Each IMM table consists of a list of all IMMs assigned to a each QP. Each IMM table is accessed through a pointer which exists in each QP table entry. Each IMM entry consists of an IMM address and the name of the relation page currently staged to that IMM.

Query Packet Wait Queue. The query packet wait queue (QPWQ) consists of a list of query requests, queued in a FIFO manner, which are waiting to be processed. Each QPWQ entry consists of an execution indicator, the query request's message number, host-id, program-id, CRT-id, and priority, and a pointer to its query packet operation list. This table is accessed whenever another query request can begin being processed by one or more QPs.

Query Packet Operation List. A query packet operation list (QPOL) exists for each of the query requests queued in the QPWQ. Each QPOL consists of an entry containing a query operation step of the query request. Each QPOL is accessed through a pointer which exists within each query request entry in the QPWQ. Each QPOL entry consists of an operation entry and a query operands entry. The QPOL is generated upon successful validation editing of the query request. It is this list which is transmitted to the QPs assigned to process this query.

Query Packet Execution List. The query packet execution list (QPEL) consists of all query requests currently being processed by one or more query processors. Each entry in the QPEL consists of a pointer to its twin entry remaining in the QPWQ (to enable access to the QPOL), the query request message number, the optimal number of QPs calculated to be used to process this query, the current number of QPs processing this query, a pointer to the query operation state table, a completion-of-processing indicator, and the name of the relation containing the answer to the query. This table is used to determine what query request to assign to an idle query processor. If all query requests currently in the QPEL are being processed by each of their optimum number of QPs, then a new query request is selected from the QPWQ, its query operation state table is created, and the idle QP is then assigned.

Query Operation State Table. A query operation state table (QOST) exists for each of the query request entries in the QPEL. Each QOST consists of a list of entries, one corresponding to each entry in the QPOL. Each entry consists of the name of the relation being processed by its corresponding query operation step, a pointer to the relation entry in the relation table (used to access its page table for QP processing), a relation page currency counter pointer, and count, total, maximum, and minimum buffers (used to gather values generated by N QPs processing COUNT, AVERAGE, MINIMUM, and MAXIMUM query operations). It is through this table that proper currency is maintained, as a multiple number of QPs process a query operation step in parallel.

Transmit Queue. The transmit queue is used to queue up, in a FIFO manner, responses to query requests for subsequent transmission back to the host computer. Responses consist of immediate answers to requests (accepted or rejected) and also answers to previously accepted requests. Each entry in the transmit queue consists of the query request's originating message number, host-id, program-id, CRT-id, and priority, the final relation name containing the answer or the immediate response answer, and a relation page transfer counter (used when the relation answer contains a multiple number of pages and may possibly require a multiple number of transmissions to the host to send the complete relation answer).

Subsystem Functions

The BCP subsystem is composed of nine specific functions: six major functions and three minor functions. The major functions are 1) check queues, 2) service a host request, 3) assign a query processor, 4) service a query processor, 5) answer a host request, and 6) backup the data base. The minor functions are 1) initialization, 2) setup, and 3) wrap up. A hierarchical tree diagram of the BCP subsystem's functions is shown in Figure 22. The discussions for each of these nine functions are presented in the following sections in a sequence which follows the diagram of Figure 22 from top down, left to right. As each of these functions are described, additional elements of this design will be brought to light.

Initialization. Initialization, a minor function, is responsible for establishing a newly created database state; a state in which the database contains no relations.

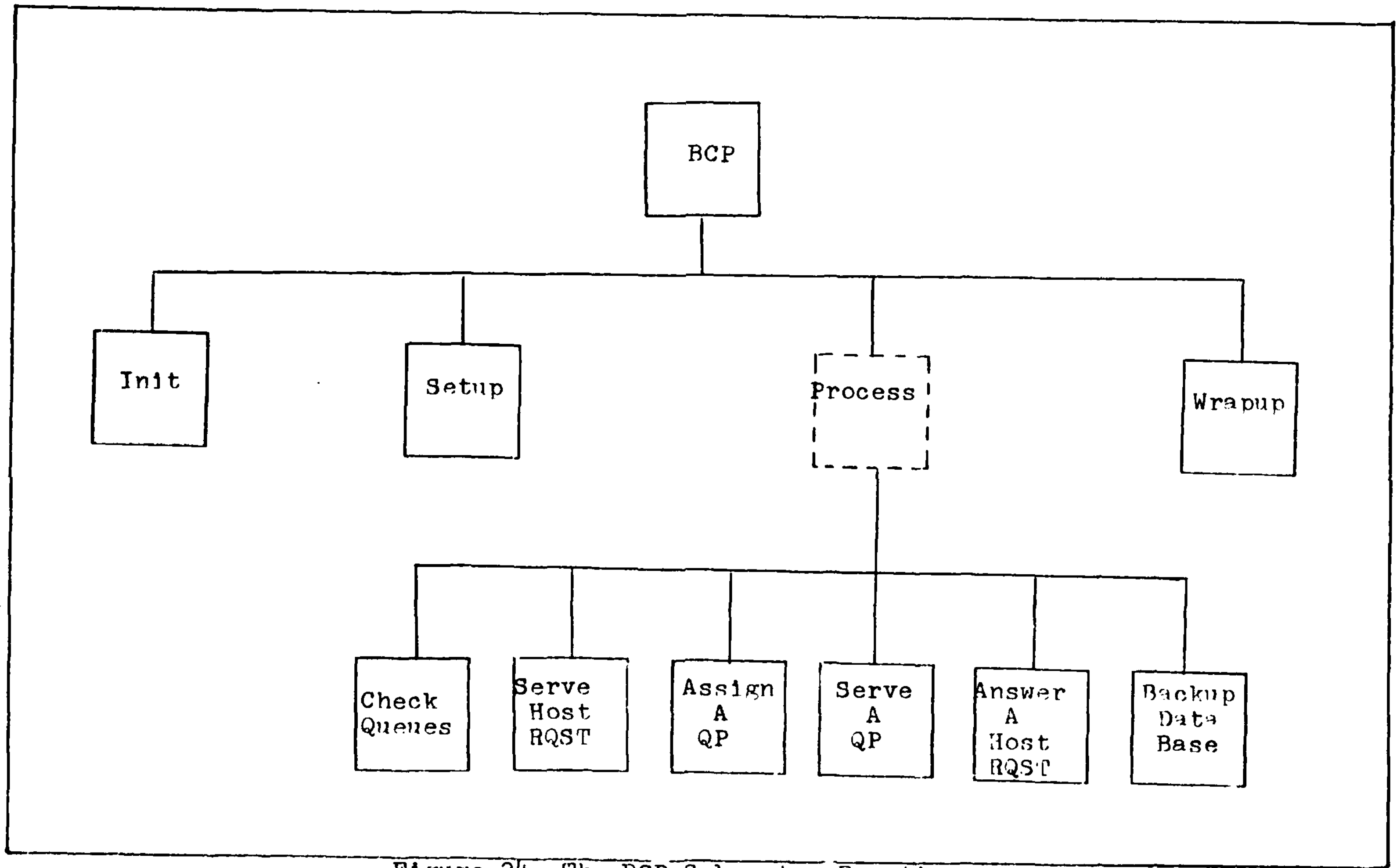


Figure 24 The BCP Subsystem Functions

At startup time for the BCP, the operator specifies whether or not a data base currently exists. When initialization is requested, the system will proceed to request the volume-id identifying where the system tables (attribute name table, relation table, relation's attribute tables, and relation's page tables) are to be saved when a save or wrapup is requested. These tables are subsequently used to accomplish the setup function the next time the system is started up. Upon receipt of a valid volume-id, a relation file and an attribute name file will be created and initialized.

Setup. Setup, a minor fuction, is responsible for establishing a state of readiness to process query requests against an already existing data base. Setup accesses the relation file, the domain name file, the attribute name file, the relation attribute file, and the relation page file from the previously specified volume-id and creates and builds the following linked lists in main memory (using dynamic memory allocation): the domain name table, the attribute name table, the relation table, the relation's attribute table, and the relation's page table (see Appendix D for more detail regarding these system files). Next the query processor table and its IMM tables are created and then configured for the current state of processing (number of QPs online for the current session). Then the volume-ids are specified, identifying where the data base relations reside. At completion of bringing the

BCP subsystem into a state of readiness, a message is sent to the host indicating a state of readiness to process query requests.

CheckQueues. Checkqueues, a major function, is responsible for the decision making as to which process function (service a host request, assign a QP, service a QP, answer a host request, backup the database, or exit to wrapup) to accomplish next. Since this function is modularized into a single PASCAL procedure, different selection algorithms can easily be substituted and used. Obviously, the process-functions 'assign a query processor' and 'service a query processor' will require much more selection service by checkqueues than the other functions since this is where the vast majority of the query processing work will occur. Special condition checks are made to ensure that 1) the query input queue does not overflow, and 2) that a database request or wrapup request is not accomplished until all the QPs become idle.

The initial approach taken for the selection algorithm was to create a predetermined selection sequence and step through this selection sequence table. The table was carefully weighted with selection entries to perform query processor assignment and query processor service functions. For each of these functions, additional checks were then made to determine if indeed this function requires processing at the instance of selection. If not, then the next entry in the selection table is accessed and checked.

Service Host Request. Service host request, a major function, is responsible for accepting a new query request from the host computer, and preparing the request for subsequent processing. By using the UCSD PASCAL intrinsics UNITREAD, UNITWRITE, and UNITBUSY, messages can be routed between processors in the background while the BCP software executes in the foreground "uninterrupted" (UCSD, 1979). This being the case, a query request will already be waiting in the BCP's input buffer when this function is called.

First the query's access right (password and security level) is verified. If a rejection occurs, a response message is built and added to the host response queue (transmit queue). If the access right is approved, then the query content is validated to ensure proper format. If the format is not correct, the request is rejected and a response message is queued. Since a query request is composed of relational algebra query operations (see Appendix B) which generate intermediate relations, as a query packet is edited, temporary entries must be created in the attribute name table, the relation table, and the new relation's attribute table in order to 1) edit successive query operations and 2) eventually process those query operations. If at any point in the editing an error is detected, all the temporary entries are deleted and a error response message is queued. When an error-free edit results, the temporary relations are permanently added to the above referenced tables. Within the relation table, the temporary (intermediate result) relations are kept track of to enable their deletion (table entries and relation pages) when they are no longer required in the processing of the query.

A new entry for this query request is then added to the query packet wait queue (QPWQ) and its query packet operation table (QPOT) is then created. A response indicating acceptance of the query request is then added to the transmit queue.

The decision was made to limit the length of a query request to 1000 bytes (size of input buffer) and also limit the length of each query operation step to 100 bytes (size of an entry in the QPOL). These size limitations can easily be modified merely by changing a single constant declaration.

Upon completion of handling this query request, the input buffer is cleared and a new UNITREAD to the host processor is issued.

Assign a Query Processor. Assign a QP, a major function, is responsible for assigning a query request packet to an idle query processor(s). When a new query packet is to be processed, the entry in the QPWQ is transfered to the QPEL and a QOST is created. At this time, an estimation is made as to the optimal number of QPs it will take to process this query. This estimate is added to the QPEL entry. Each entry in the QPEL is examined to see if another QP could be assigned to this query packet. When one is found, the number of QPs in use is updated in the chosen QPEL entry and the QPOL is then formatted into a message and sent to the selected QP. If all QPEL entries are using their full allocation of QPs, then a new query packet from the QPWQ is transferred to the QPEL. If the QPWQ is empty then a query packet in the QPEL will be assigned more than its QP allocation in order not to waste QP usage.

Service a Query Processor. Service a QP, a major function, is responsible for servicing a query processor request. Those requests to be serviced include 1) get next page, 2) get page n, 3) update page n, 4) add page, 5) destroy relation, and 6) query processing completed. Since relations consist of N fixed length pages, in order to process a query request operation for a specific relation, the QP has to process these pages one at a time.

'Get next page' requests are used in processing selects, projects, joins , unions, deletes, and max, min, ave, and count. For these operations, 'get next' rather than 'get page m' is used since a multiple number of QPs may be processing this operation in parallel.

'Get page n' requests are used in processing join and union operations (relation B of the pair of relations being either joined or unioned). In this case each QP must join all pages of relation B to the page of relation A received as a result of the 'next page' request.

'Update page n' requests are used for add, change and delete operations performed on tuples within a page of a relation previously fetched.

'Add page' is used to add a new page to a relation generated as a result of a select, project, join, or union.

'Destroy relation' requests are used to tell the BCP to delete a relation (a temporary relation in most cases) from the data base.

'Query processing completed' tells the BCP that this query packet has been completely processed and that now the QP is idle.

A response message is formatted and sent back to the QP indicating what action was taken for that request. Each time a request is serviced, the QOST relation currency indicator entry is updated. For a 'next page' and 'get page n' request, if there is another page yet to be processed, it is staged into that QP's IMM, and then a response indicating the action taken is given. For the 'update page n', 'add page', and 'destroy relation' requests, a response indicating the accomplished action is all that is necessary to be given. 'Query processing completed' messages receive no response from the BCP since the QP automatically goes into a state of wait for a new query packet assignment.

Answer a Host Request. Answer a host request, a major function, is responsible for communicating response messages back to the host computer. To ensure coordinated transmission between processors, the BCP responds with a message back to the host computer for each message request sent to the BCP. An immediate response message is sent to the host indicating acceptance or rejection of a request, and then at a later time the answer to the processed request is sent back to the host.

Backup the Data Base. Backup the data base, a major function, is responsible for saving the relation table, the attribute name table, the relation's attribute table and the relation's page tables, residing in main memory, back to the predesignated mass storage device. This function is accomplished when either a data base save request or a wrapup request is submitted.

Wrapup. Wrapup, a minor function, is responsible for calling the function "backup the data base" and then performing a shutdown of the BCP software. This function is accomplished when a wrapup request is submitted.

Subsystem Logic Flow

An overview of the system logic flow is defined in this section. It is presented in three parts, each corresponding to the processing performed by the host computer, the backend controller processor, and the query processor. Each part's logic flow is presented in a top-down tabular format. The subsystem logic flows for the host and the query processors have been included in this section to give the reader a clearer picture in understanding the subsystem logic flow for the backend controller processor.

Note that this system's logic flow is designed to accomplish all three levels of parallelism in the processing of queries: independent parallelism, pipelining, and node splitting; processes which will be easily accomplished due to the utilization of the paging of relations.

Host Computer Logic Flow

(1) If a new query has been submitted by Roth's System
  (a) convert the query into a sequence of relational operation steps
  (b) construct a query packet message
  (c) place the query packet message on the query packet wait queue within priority

(2) If an interrupt request from the BCP has occurred
  (a) receive the response message
  (b) format the response for shipment to awaiting user
  (c) ship the response to the user

Backend Controller Logic Flow

(1) If ready to begin a new processing session
  (a) if requested, perform initialization for a new data base information system
  (b) perform a setup of the database system tables

```
(2)  If ready to begin processing a new query packet
     (a) validate the user's access right; if fail, reject
         the query back to the host computer
     (b) validate the query; if unable to answer, reject
         the query back to the host computer
     (c) estimate the optimal number of QPs to be
         utilized
     (d) add the query packet to the execute table
(3)  If a query processor is available
     (a) select a query packet for processing
     (b) if not already done, create a query operation
         state table (QOST)
     (c) determine next page(s) of relation(s) to stage
     (d) if not already staged, stage page(s) to IMM(s)
     (e) if required, lock the relation
     (f) construct message to be sent to QP
     (g) update the QOST
     (h) send message to the QP
(4)  If a query processor issued an interrupt request
     (a) accept the QP's message
     (b) if an update page request
         (1) get page m address
         (2) switch to IMM address
         (3) write page back to mass storage
         (4) send update response back to QP
     (c) else if it is a next-page request
         (1) add count, total values to QOST entry
```

```
        (2) update max, min values in QOST entry
        (3) if another page can be processed by this QP
            (a) if required
                (1) switch to IMM address
                (2) stage the page to the IMM
            (b) update the QOST
            (c) send next-page response to QP
        (4) else since no more pages for this QP to
            process
            (a) update the QOST
            (b) send next-page (NUL) response to QP
    (d) else if it is a get-page m request
        (1) if another page exists to be processed
            (a) if required
                (1) switch to IMM address
                (2) stage page m to IMM
            (b) update the QOST
            (c) send the get-page m response to the QP
        (2) else since Relation B has no more pages
            (a) update the QOST
            (b) send a get-page m (NUL) response to QP
    (e) else if it is an add-page request
        (1) determine a new page address
        (2) switch to IMM address
        (3) write page to mass storage
        (4) update system tables
        (5) send add response back to QP
        (f) else if it is a destroy relation request
            (1) destroy all relation page entries
            (2) update system tables
            (3) send destroy relation response to QP
        (g) else since it is a query-completed response
            (1) update the system tables
            (2) unlock any locked relations
            (3) move query request to the response queue
            (4) set the QP into an available status
    (5) If a response is queued for transmission to host
        (a) if an immediate response type message
            (1) build message
            (2) send message to host
        (b) else since it is an answer type message
            (1) retrieve the answer relation
            (2) for each page of the relation
                (a) build the message
                (b) send message to host with indicator that
                    more of the answer is contained in the
                    next message
    (6) If a new query has arrived from the host
        (a) queue the query request in the wait queue
    (7) If a database save request has arrived
        (a) set indicator so no more requests may arrive
        (b) when all existing queries have been serviced
            (1) perform a save of all system tables
            (2) reset indicator to now allow new requests in
```

```
(8)  If a wrapup request has arrived
     (a) set indicator so no more requests may arrive
     (b) when all existing queries have been serviced and
         answers have been transmitted
         (1) perform a database save
         (2) inform host that BCP is terminating session
```

Query Processor Logic Flow

```
(1) At power up, initialize and wait for message from BCP
(2) Accept message from BCP
(3) If operation = SELECT or PROJECT
    (a) if state = initial message
        (1) initialize
        (2) switch to IMM address
        (3) SEL or PRO tuples, write into local memory
            (LM)
        (4) if LM becomes full
            (a) switch to destination IMM address
                assigned
            (b) write LM contents into IMM
            (c) request BCP to add the page
        (5) else if page processing completes
            (a) request BCP for next page to process
    (b) if state = response to add request
        (1) go to step 3a3
    (c) else since state = response to next-page request
        (1) if received a next-page address to process
```

```
                (a) switch to IMM address given
                (b) go to step 3a3
            (2) else since no more pages to process
                (a) switch to destination IMM address given
                (b) write LM contents to IMM
                (c) request BCP to add page
(4) if operation = JOIN
    (a) if state = initial message
        (1) initialize
        (2) switch to IMM address
        (3) read page of Relation A into LM
        (4) switch to IMM address given for Relation B
        (5) perform JOIN storing tuple results in LM
        (6) if LM becomes full
            (a) switch to destination IMM address
            (b) write LM contents into IMM
            (c) request BCP to add page
        (7) else if processing of page has finished
            (a) request BCP for Relation B page m+1
                IMM address
     (b) if state = response to add request
         (1) go to step 4a5
     (c) else since state = response to get-page m+1
                                of relation B request
         (1) if received a page IMM address
             (a) switch to IMM address
             (b) go to step 4a5
```

```
                (2) else since no more pages (NUL) to process
                    (a) switch to destination IMM address
                    (b) write LM contents into IMM
                    (c) request BCP to add page

            (d) else if state = response to next-page request
                (1) if received a next page address to process
                    (a) go to 4a2
                (2) else since no more pages to process
                    (a) go on to next instruction
    (5)  If operation = UNION
         (a) request next page of relation A
         (b) if get another page,
             (1) request add page to relation C
             (2) go to 5b
         (c) else since no more pages in relation A
             (1) request next page of relation B.
             (2) if get another page,
                 (a) request add page to relation C
                 (b) go to 5c1
             (3) else since no more pages in relation B
                 (a) go on to next instruction
    (6)  If operation = MODIFY or DELETE
         (a) if state = initial state
             (1) initialize
             (2) switch to IMM given
             (3) read tuples in page
            (a) if key match occ
                (1) modify or delete the tuple
            (b) go to 6a3
        (4) request update of page and next-page
    (b) else since a response to next-page
        (1) if received a page
            (a) go to 6a2
        (2) since are no more pages
            (a) go on to next instruction
(6) If operation = ADD
    (a) switch to IMM address given
    (b) search for an empty tuple position
    (c) if there is sufficient room
        (1) add the tuple to the page
        (2) request the BCP to update the page
    (d) else since sufficient room exists
        (1) add the tuple to a new page (different IMM)
        (2) request the BCP to add the new page
(7) else since operation = MIN, MAX, COUNT, or AVERAGE
    (a) if state = initial message
        (1) initialize
        (2) switch to IMM address given
        (3) read tuples and calculate MIN, MAX, COUNT,
            TOTAL
        (4) request the next-page IMM address and also
            at the same time, return current values
            calculated
```

```
(b) else since state = response to next-page request
    (1) go to step 7a2
```

Note that a query request, which desires to perform tuple adds, can only contain tuple add operations. The BCP will then generate an individual query packet for each tuple add operation. This process will ensure full utilization of all query processors and not have to be concerned with multiple adds occurring for the same tuple add operation.

Inter-processor Communication

Communication between the processors within this system consists of the following activities.

(1) Host to BCP communication transmission

(2) BCP to Host communication transmission

(3) BCP to QP communication transmission

(4) QP to BCP communication transmission

In the initial stage of the development of the backend multiple-processor relational data base computer system, parallel and serial bit transmission channel communication links are used in connecting the processors together. Data transmission through these channel links is accomplished using UCSD PASCAL intrinsics UNITREAD and UNITWRITE which reference packed array buffers containing the messages to be transmitted. This method provides a foreground / background multiprogramming environment resulting in a savings for the software residing within each processor of not having to perform programmed I/O for data transmission.

<u>Host / BCP communication</u>. Two packed array buffers are reserved in both the host and the BCP software, one for sending requests to the BCP / receiving requests from the host, and another for receiving responses from the BCP / sending responses to the host. By having a paired set of communication buffers, and using the UNITREAD, UNITWRITE, UNITBUSY, and UNITWAIT instructions within UCSD PASCAL, unsupervised I/O can simultaneously occur between the host and BCP.

<u>BCP / QP communication</u>. Since the QPs are slaves to the BCP, they must wait for response messages to their requests sent to the BCP before continuing their processing. Thus, only one communication buffer is required within each QP's software to perform I/O with the BCP. Since the BCP is the master to a multiple number of QPs, it must maintain a separate communication buffer for each QP with which it communicates. The BCP uses the UNITREAD, UNITWRITE and UNITBUSY instructions when communicating with the QPs. A UNITREAD instruction is issued for each of the busy QPs in anticipation of an upcoming QP request. When Checkqueues so dictates, a check is then made (using the UNITBUSY instruction) to see if the UNITREAD for a particular QP has been satisfied. If the request was satisfied, then the request is processed, a response using a UNITWRITE is made (with wait), and then

a new UNITREAD is accomplished. If the request is still outstanding, then the next QP's corresponding UNITREAD state is checked.

The specific message formats for communication from the BCP to the QPs, and from the QPs back to the BCP are contained in Appendix C. A set of comments appear with each communication format for eack type of QP request and BCP response.

Summary

Within this chapter, the detailed design of th backend controller processor subsystem and its interprocessor interfaces have been defined. Defined and described were the subsystem functions, the subsystem data tables necessary to support the BCP functions, the subsystem logic flow, the query operation specifications and, finally, the interprocessor communication details. With this information defined in detail, a top down structured modular design of the BCP subsystem was then accomplished. This design, shown in structure chart form in Appendix A, was then coded into UCSD PASCAL program code, each structure chart module corresponding to a PASCAL procedure (subroutine). Before any testing of the BCP software took place, the entire BCP subsystem design had been completely coded and thoroughly reviewed.

The next chapter will now address the planning and accomplishment of the implementation and testing of the BCP subsystem, the heart of this backend computer system.

V. Subsystem Implementation and Testing

Introduction

This chapter covers the implementation and testing of the software and inter-processor communication hardware interfaces for the backend controller processor subsystem of the backend multiple-processor relational data base computer system. In the first section, the approach chosen for accomplishing the implementation and testing of the BCP subsystem is presented. Resources and constraints directly affecting the implementation and testing are then identified. In the next section the support software, designed and implemented for the purpose of supporting the implementation and testing of the BCP subsystem software, is identified and presented. The final section presents the implementation and testing results and then gives an evaluation summary.

The fourth and final goal of this thesis project was to fully implement the BCP subsystem, within a first stage environment, as specified in the system development plan identified in Chapter III. As a result of underestimating the overall size of the BCP subsystem, a partial implementation and testing could only be completed within the time frame given for this thesis effort. Within the content of this chapter, qualification will be given of what was planned and indeed accomplished verses what was planned but could not be accomplished.

Implementation and Test Approach

Once the BCP subsystem's software was fully designed, pseudo-coded, and programmed in the UCSD PASCAL language, the next step taken was to create an implementation and test plan. Within this plan, several activities for the total implementation effort were identified. These activities are listed below.

(1) Using a top down approach strategy for testing, create a module test plan specifying the sequence of tests anticipated. If top-level modules are implemented before lower-level modules, the need for driver programs is eliminated, and major interface problems are exposed before they affect the logic of lower-level modules.

(2) Perform an incremental top down testing of the program modules using program "stubs". Incremental testing is an essential aspect of the top down testing philosophy. Instead of unit-testing a large number of modules and linking/compiling them together in one test (only to discover that the system does not work and the error is hard to find), systems should be tested in a more controlled fashion. A small, high-level subset of the system should be tested until it works. Since the high-level subset calls lower-level modules, the lower-level modules should be simulated using program "stubs" containing real module linkage logic, but not actually performing any of the lower-level detailed work. Program

stubs could return a message or a constant output, or simulate the timing of the lower-level work before exitting. Once a high-level subset of the system is working, testing continues by substituting program stubs one at a time until the last module of the system has been added.

(3) Create the test data for each module/function based on the following module testing approaches.

(A) Insure that all program segments of code are executed. This is done by choosing test data to cause each decision branch to be taken. Also, if a branch is dependent upon a compound logical expression, test data must be chosen for each of the possible logical combinations.

(B) Insure that all equivalence classes for input data are tested. This procedure entails breaking the input into its components and testing combinations of valid components until all equivalence classes of valid components have been converted. Then the invalid components from each equivalence class are tested individually.

(C) Insure that all boundary values are tested. This procedure entails testing the values at the borders of the equivalence classes to detect "off by one" errors.

The test plan for stage one of this project was subdivided into seven phases of testing. The creation of a seven phase test plan was necessary since a multiple number of independent processors, each containing its own subsystem software (developed software or test software), were required to fully test the BCP subsystem. The seven phases of software testing consist of the following.

(1) Test the newly created operating systems, for each processor, containing the channel driver software for interfacing multiple processors together. (Appendix F contains all the necessary information to accomplish the task of creating these new operating systems).

(2) Test the front-end host software in preparation for interfacing the front-end and BCP processors.

(3) Test the BCP software up to a point where interfacing to the query processors is required for further testing.

(4) Test the query processor software in preparation for interfacing the BCP and query processors.

(5) Test the BCP software up to a point where interfacing to the IMMs is required for further testing.

(6) Test the IMM test software in preparation for interfacing the BCP and IMM, and the QPs and IMM.

(7) Complete testing of the BCP subsystem.

One factor that significantly increased the implementation effort was the need for complete error and detection recovery as earlier described. Every module had to be protected to withstand any user input regardless of its likelihood. This is because a process, in any one of the backend's processors, aborting due to an out-of-range variable or other invalid input could result in a contamination of the data base. Therefore, every module was implementated to validate the inputs first and issue error messages to be transmitted back to the user in lieu of the normal response if an invalid input was detected. This greatly increased the size and complexity of the modules and yet did succeed in making the modules impervious to invalid user inputs.

Resources and Constraints

The specific hardware configuration used to accomplish the implementation and testing of the BCP software is shown in Figure 25. Five processors were determined to be necessary and sufficient to fully accomplish the Stage I testing. Since dual-port IMMs were not available at the time of this stage of development, a processor was used to simulate the IMM. Appendix E presents the specific hardware interface details. Future upgrading in the follow-on stages of this development effort are outlined in the system development plan section of Chapter III.

Constraints encountered or foreseen in this stage of the development consist of the following items.

(1) Each time a modification (correction, modification, or module addition) is made, the entire subsystem's software package must be recompiled. Compiles currently take 15 minutes to accomplish, thus making developmental testing a slow process. Compile times are expected to take longer as

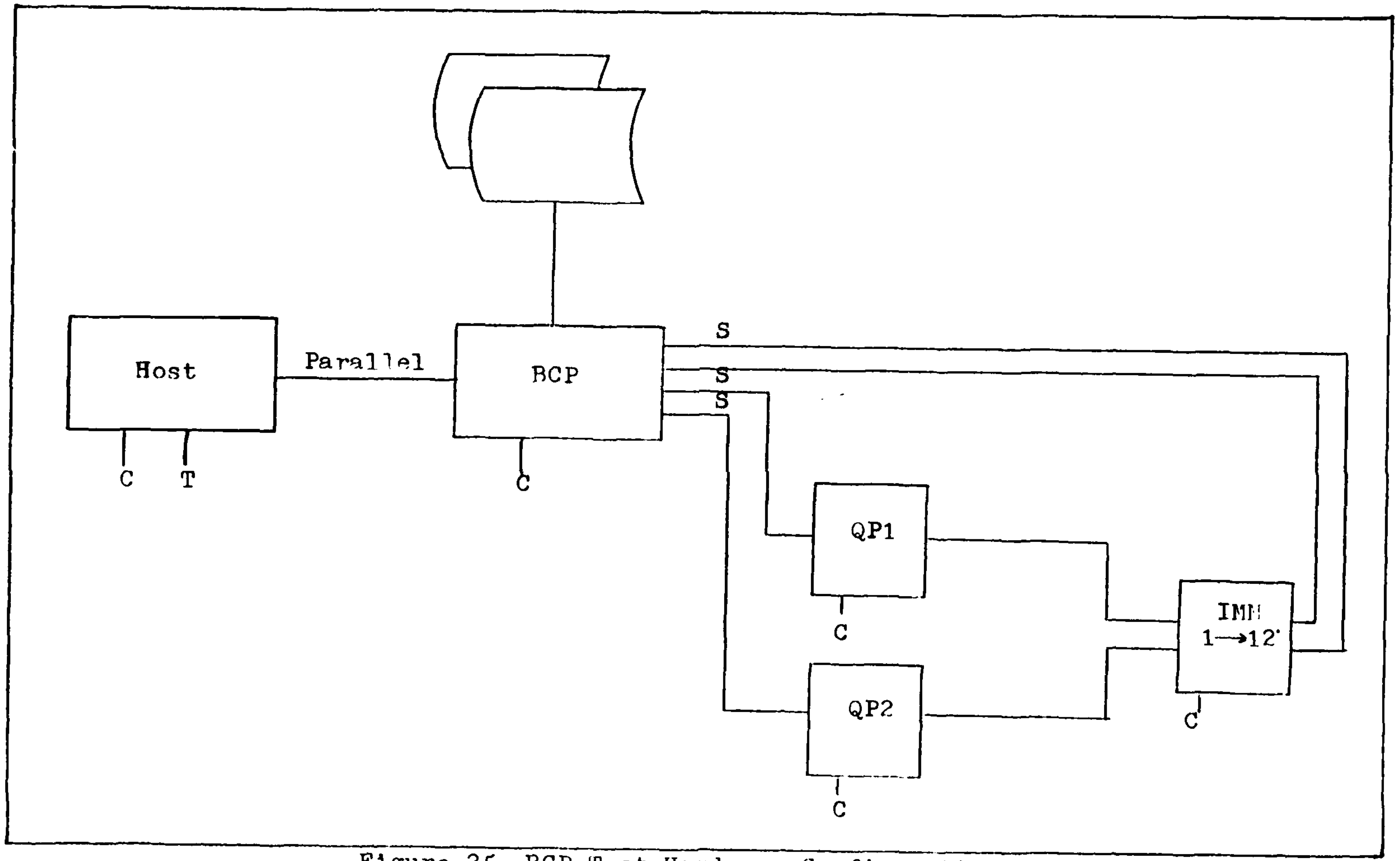


Figure 25 BCP Test Hardware Configuration

more and more modules are added to the subsystem.

(2) The UCSD PASCAL system, while allowing for the dynamic allocation of memory, does not have a mechanism to enable the release of dynamically allocated memory. Since the BCP subsystem makes heavy usage of temporary table entries (for the validation editing of new query requests and for the creation of temporary "intermediate result relations" table entries), its own internal memory management system will have to be developed.

(3) The UCSD PASCAL system limits the number of UNITNUMBERS to just 12 entries; a number sufficient to interface, (in addition to the host and IMM), only two query processors to the BCP. The operating system will require modification before more query processors can be added.

(4) Only 56 K bytes of memory are available on an LSI-11/02 in which to contain the BCP software, all its interface communication buffers, the space for the dynamically growing and shrinking subsystem tables, and the driver software (one copy per UNITNUMBER). More memory will be required as the configuration and/or data base is expanded; snould consider going to a LSI-11/23 computer.

Test Support Software

To fully test the BCP subsystem requires the development of a set of "processor test support software stubs". Specifically, software stubs had to be written to test the front-end host processor interface, the query processors interfaces, and the IMM (processor simulated) interfaces. Each are described below.

Host Processor Software. The host processor software, (designed, implemented and tested within this thesis effort), was required in order to test the query request / query response interfaces between the front-end host processor and the BCP. Inputs generated by this test package and sent to the BCP consist of the following listed items.

(1) A sign-on request

(2) A relational query request

(3) A data base save request

(4) A wrapup request

Through the host's console, a text command is entered to specify which of the above actions to accomplish. For query requests, the operator simply specifies a pre-created query disk file as the request to be submitted. Outputs received from the BCP (immediate responses or query answer responses) are displayed on the operator's console. Answer responses containing multiple message blocks are easily handled by the host software using a predefined protocol. Through the use of UNITREAD and UNITBUSY instructions, all I/O communication overhead for the host and BCP is eliminated. The front-end host software design is shown in structure chart form in Appendix G. Its program code is contained in Volume II of this thesis.

Query Processor Software. The query processor software (conceived and shown in structure chart form in

Appendix H), not yet designed in detail, would be required to test the BCP / QP interface in which the BCP assigns query packets to a QP, receives query operation requests, performs internal answer formulation and actions, and supplies responses to the QP(s). Software would be required to accept the query packet messages and display them on the QP's console for operator inspection, accept and transmit operator formulated QP requests, and then accept and display the BCP responses to those requests. In this way, the QP dependent functions accomplished by the BCP for staging and updating of relations to the QP's IMMs could be fully tested.

IMM Software. The IMM, simulated by a processor whose software design has not yet been formulated, would be required to test the BCP / IMM interface and the QP / IMM interface. Software would be required to accept commands from both the BCP and the QP to accomplish receiving / supplying relation pages. The commands are necessary since direct addressing to the IMM cannot yet be accomplished. Due to a shortage of UNITNUMBERS, only one IMM could be included in this configuration, so both QPs had to be supported by a single IMM. For the full implementation, multiple IMMs will be assigned to each query processor.

Test Results and Evaluation

Within the BCP software, the following functions were completely developed and tested.

(1) Initialization and Setup of the database system.

(2) Receipt of query requests from the host.

(3) Validation of query requests.

(4) Message responses to the host.

(5) Assignment of query requests to QPs.

(6) Performance of data base saves.

(7) Performance of wrapup and shutdown.

For that testing which could be accomplished in the available time, no problems were encountered. The detailed design has proven, thus far, to be very complete, having no inconsistencies or failures. Continued development and testing of this subsystem should prove to be a smoothly accomplishable task.

IV. Conclusions and Recommendations

Overview

At the outset of this investigation, four problems / goals were identified. The first problem was to determine the feasibility of applying multiple-processor techniques to the implementation of a relational DBMS within a micro/mini-computer system environment. The second problem was to determine both the long range and short range system requirements and goals. The third goal was to develop a structured, modular system design (software and hardware using state-of-the-art technology) for implementing the reqired system over the longterm, identifying experimental tradeoffs. The fourth and final goal was to implement a first stage system model to show feasibility and to investigate tradeoff alternatives.

The first problem was met to a great extent based on the information brought to light within the background chapter (Chapter II) of this thesis. Through the merging of three relatively new concepts (backend data base computer systems, the relational data model, and data base computers having specialized architectures), the feasibility of applying multiple-processor techniques to the implementation of a relational DBMS within a mini/micro-computer system environment became a certainty.

The second problem was clearly defined and addressed in the system development chapter (Chapter III) of this thesis. The longterm requirements and goals for the development of this data base computer system were specifically defined, drawing from the data gathered in the feasibility study the best approaches in each of the three researched areas. A system design, at the overview level, was next accomplished. The system functions and interrelationships, the system hardware architecture, the system support software, and the architectural alternatives were defined. Due to the complexity and size of this system, a system development plan was then developed in order to effectively organize the development of this system over several thesis research efforts, into its final configuration.

The third goal was also clearly accomplished, and is addressed in the subsystem design chapter (Chapter IV) of this thesis. A fully detailed design of the BCP subsystem and its interfaces to the front-end , query processors, and IMMs was accomplished.

The fourth and final goal was, due to a time shortage, only partially accomplished.

Recommendations

As specified in the system development plan section of Chapter III, this thesis effort is only the initial stage of a multiple stage effort to construct a full state-of-the-art implementation of this backend multiple-processor relational data base computer system.

Because of the modular design of this system's architecture, several of the follow-on stages to this implementation can be accomplished through concurrent research efforts.

There is a large amount of very interesting research and development to be pursued in this area of computer science. The following recommendations are given concerning what specifically needs further work/investigation.

(1) Expand the front-end host into a real front-end. Perhaps design and implement modifications to Roth's relational data base system (Roth, 1979).

(2) Interface the front-end to the AFIT digital engineering laboratory computer network for the subsequent implementation of a distributed data base system.

(3) Improve the mass storage technology from floppy disk to moving-head or fixed-head hard disk devices. Establish DMA linkages from mass storage directly to the IMMs and also to the front-end computer. Then offload all staging and front-end communication to a separate processor (a little brother to the BCP).

(4) Implement the IMMs using bubble memory, CCD memory, hard disk associative memory, or integrated circuit associative memory within either a multi-port memory access or cross-point matrix access configuration.

(5) Transition the entire system to an alternative system architecture. Consider implementing all the processors on a single or set of common busses. Consider using the new Intell 432 micro-processor.

(6) Implement a full backup and recovery capability for the system.

(7) Implement a distributed data base system, storing databases at multiple computer sites, using either the DEL network, a mini-network of micro-computer systems, or the ARPANET or AUTODIN II.

(8) Establish a set of performance evaluation tools for measuring, fine-tuning, and comparing alternative configuration implementations.

(9) Consider transitioning from PASCAL to an alternative high-order language such as PL/Z, C, or ADA.

(10) Consider alternative data storage techniques and access techniques to the "sequential tuple access within a page / sequential page access within a relation" techniques.

Final Comment

Backend multiple-processor data base computer systems using both the relational data model and state-of-the-art associative storage devices have great potential for revolutionizing the information industry. By now solving the efficiency problem, using multiple-processors and state-of-the-art associative storage devices, the relational view promises a simple, flexible approach to the industry's information retrieval problem. Incredible effort has been expended on the improvement of software efficiency for DBMSs based, unfortunately, on conventional computer architectures. With the advent of state-of-the-art associative storage devices, and the use of multiple micro-processors, it is now time to channel some of this effort where even larger improvements can now be attained; improvements just no longer reachable through software improvements.

Bibliography

1. Banerjee, Jayanta and David K. Hsiao, "DBC - A Database Computer for Very Large Databases," IEEE Transactions on Computers,C-28(6):414-429 (June 1979).

2. Boral, Haran and David J. DeWitt, Implementation of the Database Machine DIRECT, Mathematics Research Center, Computer Sciences Technical Report #442, University of Wisconsin, Madison, August 1981.

3. Bray, Olin H. and Harvey A. Freeman, Data Base Computers, Lexington, D.C. Heath and Co., 1979.

4. Canaday, R.E. et al. "A Backend Computer for Data Base Management," Communications ACM 17, 10 (Oct 1974): 575 - 582.

5. Chang, Philip Y.,"Parallel Processing and Data Driven Implementation of a Relational Data Base System," Proceedings of the ACM:314-318 (Oct 1978).

6. Date, C.J., An Introduction to Database Systems (Second Edition), Reading: Addison-Wesley, 1977.

7. DeWitt, David J., "DIRECT - A multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, C-28,(6):395-406 (June 1979).

8. Hutchison, J.S. and W.G. Roman,"MADMAN Machine," Computer Architecture News,7,(2) , (August 1978).

9. Maryanski, Fred J., "Backend Database Systems," Computing Surveys, 12 (1):3-25 (March 1980).

10. Maryanski, Fred J. and V.E. Wallentine, "A Simulation model of a backend DBMS," Pittsburgh Modeling and Simulation Conference, April 1976 : 243 - 248.

11. Ozkarahan, E.A. and K.C. Sevcik,"Analysis of Architectural Features for Enhancing the Performance of a Database Machine," ACM Transactions on Database Systems,2,(4):297- 316 (Dec 1977).

12. Ozkarahan, E.A. et al.,"RAP - An Associative Processor for Data Base Management," AFIPS Conference Proceedings, 44:379- 387 (1975).

13. Ozkarahan, E.A. et al.,"Performance Evaluation of a Relational Associative Processor," ACM Transactions on Database Systems,2,(2):175-195 (June 1977).

14. Roth, Mark A., The Design and Implementation of a Pedagogical Relational Database System, Masters Thesis, Air Force Institute of Technology, Dayton, Ohio, 1979.

15. Schuster S. and E.A. Ozkarahan,"A Virtual Memory System for a Relational Associative Processor," AFIPS Conference Proceedings,45:855-862 (1976).

16. Schuster, Steward A. et al.,"RAP.2 - An Associative Processor for Databases and its Applications," IEEE Transactions on Computers, C-28,(6):446-458 (June 1979).

17. Su, Stanley Y.W. et al.,"The Architectural Fearutes and Implementation Techniques of the Multicell CASSM," IEEE Transactions on Computers,c-28,(6):430-445 (June 1979).

18. UCSD (Mini-Micro Computer) PASCAL, Version II.0 Institute for Information Systems, University of California, San Diago (March 1979). (Available from AFIT/ENE).

Additional Readings

1. Adiba, M. et al.,"Issues in Distributed Data Base Management Systems: A Technical Overview," Issues in Data Base Management, Proceedings of the Fourth International Conference on Very Large Data Bases:127-153 (1978).

2. Anderson, Donald R.,"Data Base Processor Technology," AFIPS Conference Proceedings,45:811-818, (June 1976).

3. Anderson, G.A. and E.D. Jensen,"Computer Interconnection Structures: Taxonomy, Characteristics and Examples," ACM Computing Surveys,7,(4):197-213 (Dec 1975).

4. Artwick, Bruce A., Microcomputer Interfacing, Englewood Cliffs, Prentice-Hall Inc., 1980.

5. Astrahan, M.M. et al.,"System R: Relational Approach to Database Management," Transactions on Data Base Systems, 1(6):93-133 (June 1976).

6. Babb, E.,"Implementing a Relational Database by Means of Specialized Hardware," ACM Transactions on Database Systems,4,(1):1-29 (March 1979).

7. Banerjee, Jayanta and David K. Hsiao, "Concepts and Capabilities of a Database Computer," Transactions on Data Base Systems,3,(4):281-318 (Dec 1978).

8. Baum, Richard I. and David K. Hsiao,"Database Computers - A Step Towards Data Utilities," IEEE Transactions on Computers, C-25,(12):1254-1259 (Dec 1976).

9. Boral, Haran and David J. DeWitt, Design Considerations for Data Flow Database Machines, Mathematics Research Center, University of Wisconsin, Madison, 1980 (AD A086374).

10. Bray, Olin and Kenneth J. Thurber,"What's Happening with Data Base Processors?," Datamation,25,(1)146-156 (Jan 1979).

11. Chamberlin, D.D.,"Relational Data-Base Management Systems," ACM Computing Surveys,8,(1):43-66 (March 1976).

12. Champine, George A.,"Four Approaches to a Data Base Computer," Datamation,24,(13):101-106 (Dec 1978).

13. Colon, Fernando C.,"Coupling Small Computers for Performance Enhancement," AFIPS Conference Proceedings,45:755-764, (1976).

14. Cullinane, J. et al., Commercial Data Management Processor Study, Cullinane Corporation, Wellesley,1975,(AD A035790).

15. DLV11-J User's Guide, (4 Channel Asynchronous SLU Interface), Digital Engineering Corporation, 1978.

16. Enslow, Philip H. Jr., Multiprocessors and Parallel Processing, New York, John Wiley and Sons, 1974.

17. Foster, Caxton C., Content Addressable Parallel Processors, New York, Van Nostrand Reinhold, 1976.

18. Freeman, Harvey A., "A Bibliography of Local Computer Network Architectures," Computer Architecture News,7,(5):22- 27 (Feb 1979).

19. Hayes, John P., Computer Architecture and Organization, New York, McGraw-Hill, 1978.

20. Hobart, William C. Jr., Design of a Local Computer Network for the Air Force Institute of Technology Digital Engineering Laboratory, Masters Thesis, Air Force Institute of Technology, Dayton, Ohio, 1981.

21. Holland, Robert H., "Improve Information Access with the Database Machine," Data Communications: 95-101 (March 1980).

22. Hutt, A.T.F., A Relational Data Base Management System, Chichester: John Wiley and Sons, 1979.

23. Katzan, Harry Jr., An Introduction to Distributed Data Processing, New York, A Petrocelli Book, 1978.

24. Kim, Won,"Relational Database Systems," ACM Computing Surveys,11,(3):185-211 (Sept 1979).

25. Lin, Chyuan Shiun, et al.,"The Design of a Rotating Associative Memory for Relational Database Applications," ACM Transactions on Database Systems,1,(10:53-65 (March 1976).

26. Liuzzi Raymond A., The Specification of a Data Base MAchine Architecture Development Faciliy and Methodology for Developing Special Purpose Function Architectures, Rome Air Development Center, 1980 (AD A090826).

27. Lorin, Harold, Parallelism in Hardware and Software: Real and Apparent Concurrency, Englewood Cliffs, Prentice-Hall Inc., 1972.

28. LSI-11 / PDP-1103 Processor Handbook, Digital Engineering Corporation, 1975.

29. Martin, James, Computer Data-Base Organization (Second Edition), Englewood Cliffs: Prentice-Hall, 1977.

30. Meldman, Monte Jay, et al., RISS: A Relational Data Base Management System for Minicomputers, New York: Van Nostrand Reinhold Company, 1978.

31. Microcomputer Interfaces Handbook, Digital Equipment Corporation, 1980.

32. Microcomputers and Memories, Digital Equipment Corporation, 1981.

33. Mohan, C.,"An Overview of Recent Data Base Research," Data Base,10,(2):3-24 (Fall 1978).

34. Muklopadhyay, Amar,"Hardware Algorithms for Nonnumeric Computation," IEEE Transactions on Computers,C-28,(6):284- 394 (June 1979).

35. Oliver, Ellen Jane, RELACS, An Associative Computer Architecture to Support a Relational Data Model, Doctoral Dissertation, Syracuse University, 1979.

36. Sehan, Amrun and Timbul Maruap Sihombing, Data Base Management System for Minicomputers, Masters Thesis, Naval Postgraduate School, Monterey, California, 1979.

37. Selinger, P. Griffith et al., Access Path Selection in a Relational Database Management System, Research Report, IBM Research Laboratory, San Jose, 1979.

38. Thurber, Kennith J., "Computer Communication Techniques," Computer Architecture News,7,(3):7-16 (October 1978).

39. Ullman, Jeffrey D., Principles of Database Systems, Rockville, Computer Science Press, Inc., 1980.

40. Wallentine, Virgil E., Project Report For Functionally Distributed Computer Systems Development : Software and Systems Structure, Department of Computer Science, Kansas State University, Manhattan, Kansas, 1977 (AD A052751).

41. Weitzman, Cay, Distributed Micro/Minicomputer Systems - Structure, Implementation, and Application, Englewood Cliffs, Prentice-Hall Inc., 1980.

42. Yau, S.S. and H.S. Fung,"Associative Processor Architecture - A Survey," Computing Surveys,9,(1):3-27 (Mar 1977).

Appendix A

BCP Structure Chart Documentation

This appendix consists of two sections documenting the modular design of the backend controller processor's software. The first section is a list identifying all the modules in the software design. The '*' that appears to the right of the module's title identifies that module as a 'common' module called by more than one calling module. The second section is a set of structure charts showing the interrelated structure of the modules within the subsystem. Note that the filled-in bottom right corner of the module box indicates that it is a 'common' module. The module numbers in each section are used as a cross-reference between both documentation sections.

BCP Module List

1.0	Backend Control Processor
1.1	Initialization
1.2	Setup
1.2.1	Build Domain Name Table
1.2.2	Build Attribute Name Table
1.2.3	Build Relation Table
1.2.4	Build Attribute Tables
1.2.5	Build Page Tables
1.2.6	Build QP Table

1.2.7	Build IMM Tables	
1.2.8	Setup Configuration	
1.2.9	Contact Host Computer	
1.3	Process	
1.3.1	Check Queues	
1.3.2	Service QP Request	
1.3.2.1	Access QP Request Queue	
1.3.2.2	Read QP message	
1.3.2.3	Process "Next-Page" Request	
1.3.2.3.1	Determine Next-Page	
1.3.2.3.1.1	Increment String Value	
1.3.2.3.2	Switch to IMM Address	*
1.3.2.3.3	Stage Page to IMM	*
1.3.2.3.4	Build Response Message	*
1.3.2.3.5	Update System Tables	*
1.3.2.3.6	Send Message to QP	*
1.3.2.3.7	Reset for next QP Request	*
1.3.2.4	Process "Get Page M" Request	
1.3.2.4.1	Get Page M Address	*
1.3.2.5	Process "Update Page M" Request	
1.3.2.5.1	Write Page to Mass Storage	*
1.3.2.6	Process "Add Page" Request	
1.3.2.6.1	Add a New Page Address	
1.3.2.7	Process "Destroy Relation" Request	
1.3.2.8	Process "Process-Complete" Response	
1.3.3	Service Host Request	
1.3.3.1	Read Host Request	

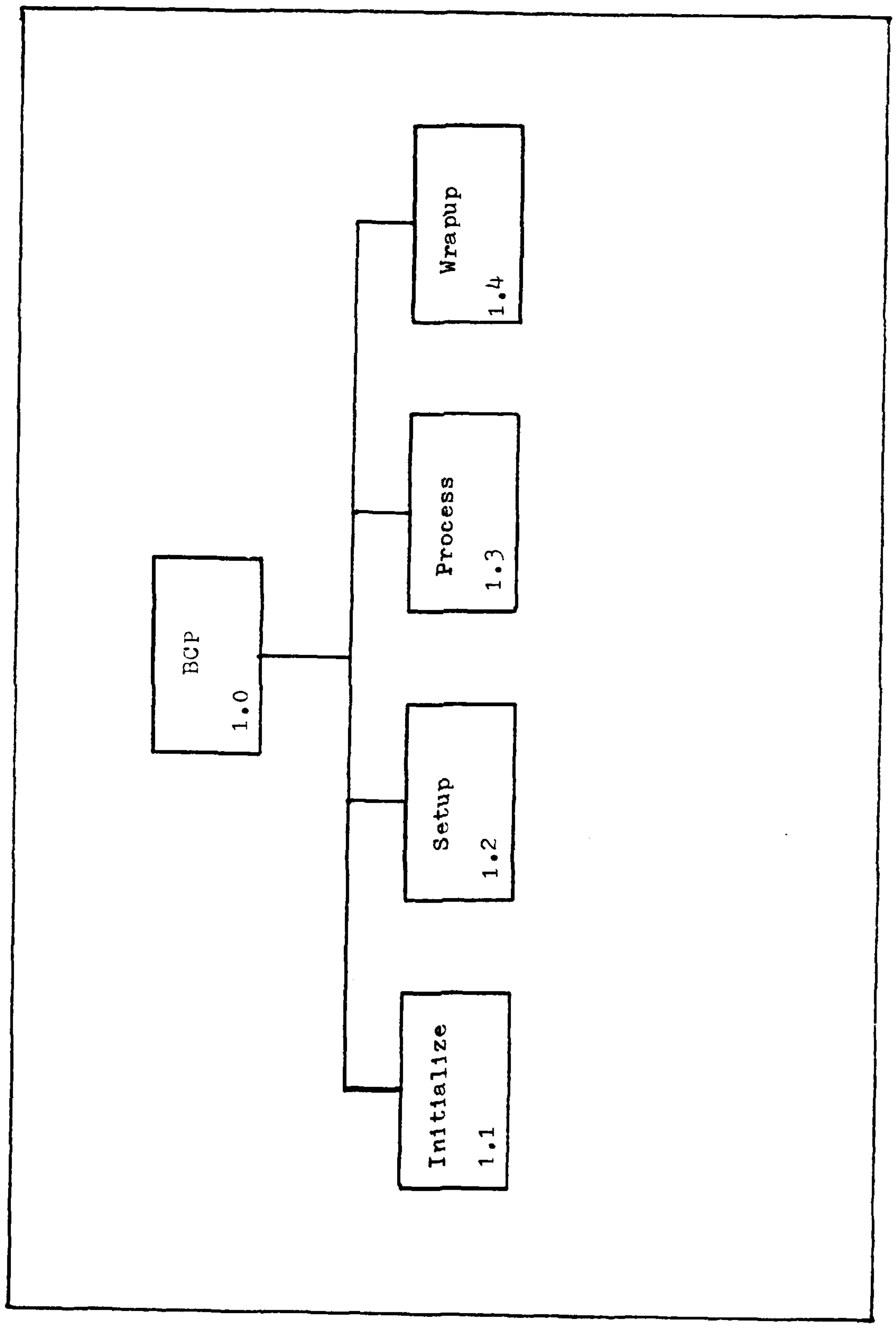
1.3.3.2	Validate User Access Right	
1.3.3.3	Validate Query Contents	
1.3.3.3.1	Validate Operation Command	
1.3.3.3.2	Validate operation Format	
1.3.3.3.2.1	Increment KK Counter	*
1.3.3.3.2.2	Scan Operand Field	
1.3.3.3.2.3	Validate Comparators	
1.3.3.3.2.4	Validate Boolean Operators	
1.3.3.3.3	Validate Relation Name	
1.3.3.3.3.1	Position to Next $ delimiter	*
1.3.3.3.3.1.1	Increment K1 Counter	
1.3.3.3.3.2	Compare with Relation Table	
1.3.3.3.3.2.1	Compare with Relation Entry	
1.3.3.3.4	Validate Attribute Membership	
1.3.3.3.4.1	Compare with Attribute Table	
1.3.3.3.4.1.1	Compare with Attribute Entry	
1.3.3.3.4.1.2	Validate Attribute Membership	
1.3.3.3.4.1.2.1	Validate Attribute Entry	
1.3.3.3.4.1.3	Validate Constant Field	
1.3.3.3.5	Create Temporary Rel Table Entries	
1.3.3.3.5.1	Save Off Relations	
1.3.3.3.5.2	Build Temporary Relation Table	
1.3.3.3.5.2.1	Find Unused Relation Code	
1.3.3.3.5.2.1.1	Check Temporary Relation Table	
1.3.3.3.5.3	Build Temporary Attr-Name Table	
1.3.3.3.5.4	Build Temporary Rel-Attr Table	
1.3.3.3.5.4.1	Find Attribute Match	

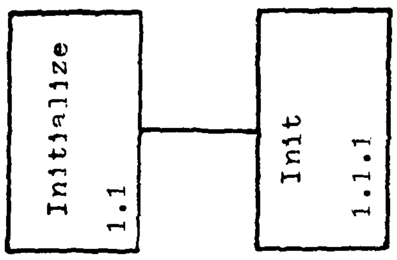
1.3.3.3.5.4.1.1	Build Attribute Work List
1.3.3.3.5.4.2	Delete Attribute Work List
1.3.3.3.5.4.3	Add Bytes
1.3.3.3.6	Add Temporary Table Entries
1.3.3.3.7	Delete Temporary Table Entries
1.3.3.4	Add QPKT to QPWQ
1.3.3.5	Build a QPOL
1.3.3.5.1	Prepare the Operation Step
1.3.3.5.1.1	Hold the Relation
1.3.3.5.1.2	Load the Relation
1.3.3.5.1.3	Load through $ Delimiter
1.3.3.5.1.4	Hold the Attribute
1.3.3.5.1.5	Convert Attribute to Offset,Length
1.3.3.5.1.6	Load the Attribute
1.3.3.5.2	Insert the Destroy Operation Steps
1.3.3.5.2.1	Add Operand to Operation Entry
1.3.3.6	Add QPKT to XMIT Queue
1.3.4	Assign a QP
1.3.4.1	Determine if a QP is Available
1.3.4.2	Dtrmn if QPKT in QPEL needs a QP
1.3.4.3	Dtrmn if QPKT in QPWQ awaiting service
1.3.4.4	Update the QPEL entry
1.3.4.4.1	Estimate Optimal QP Usage
1.3.4.4.2	Create the QOST
1.3.4.5	Build Initial Message for QP
1.3.4.6	Send Initial Message to QP
1.3.4.7	Move QPKT from QPWQ to QPEL

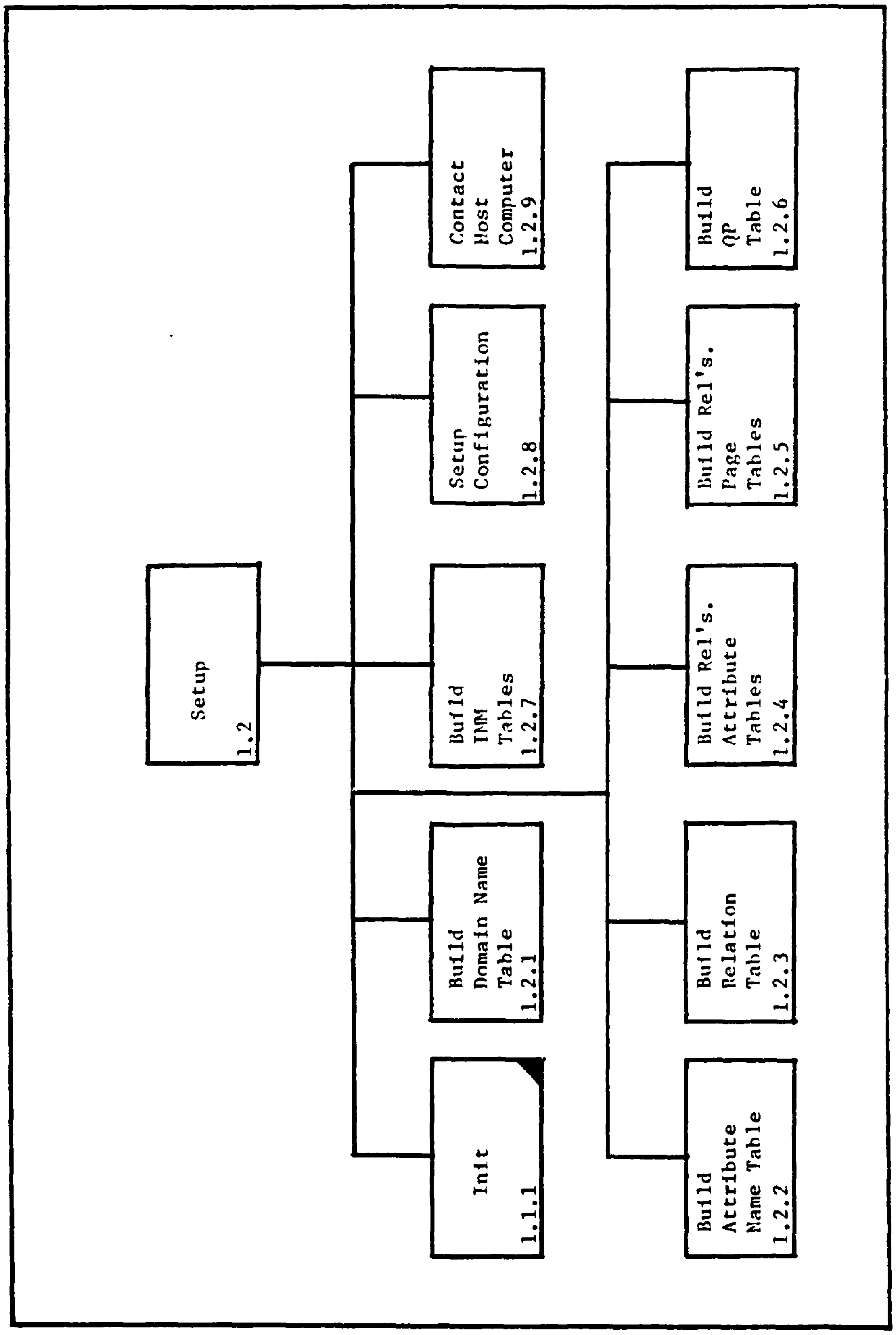
1.3.5	Transmit to Host	
1.3.5.1	Load Relation's Attribute Description	
1.3.5.2	Load Relation's Tuples	
1.3.6	Backup the Relational Data Base	
1.3.6.1	Save the Domain Name Table	*
1.3.6.2	Save the Attribute Name Table	*
1.3.6.3	Save the Relation Table	*
1.3.6.4	Save the Rel's Attribute Tables	*
1.3.6.5	Save the Rel's Page Tables	*
1.4	Wrapup	

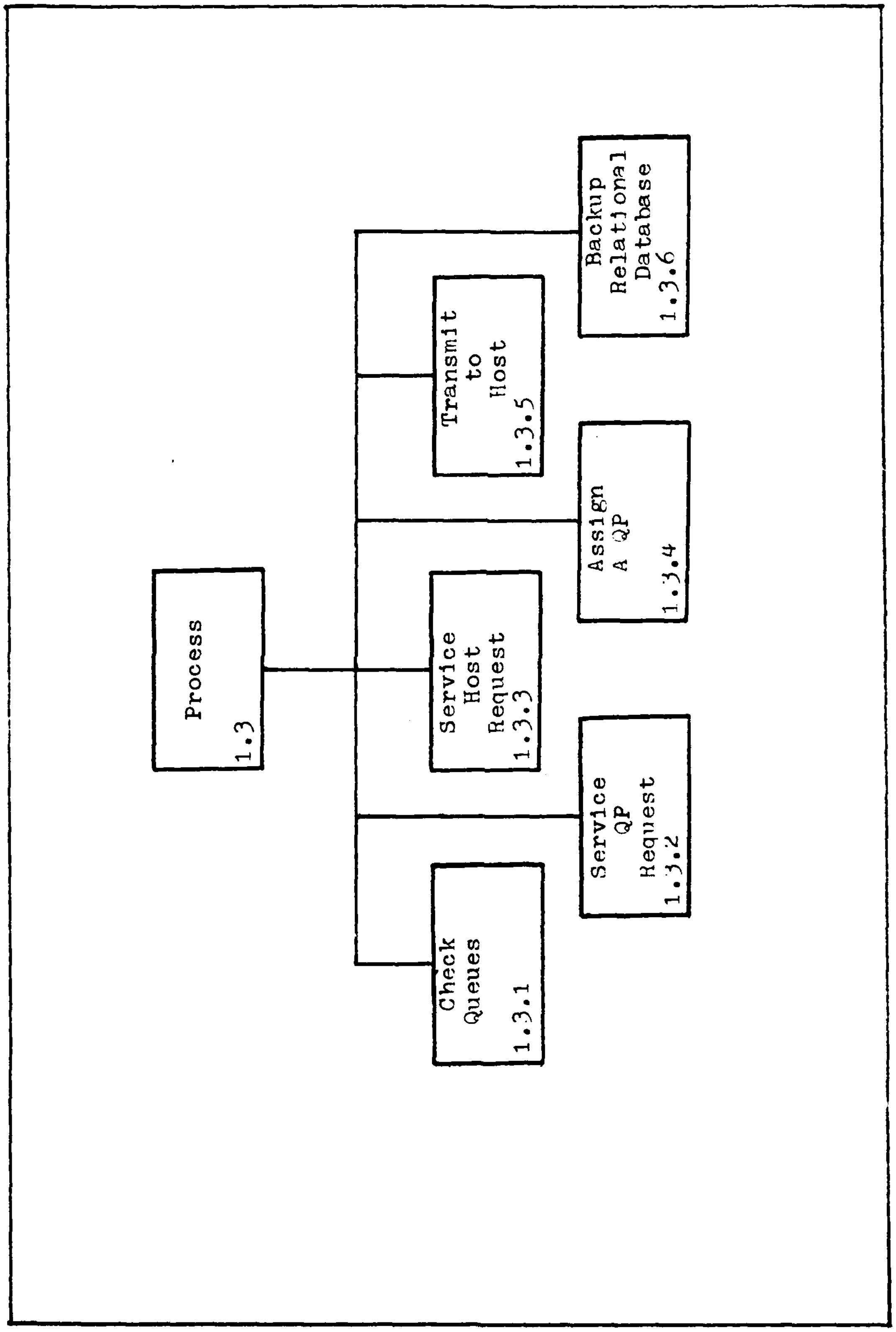
BCP Structure Charts

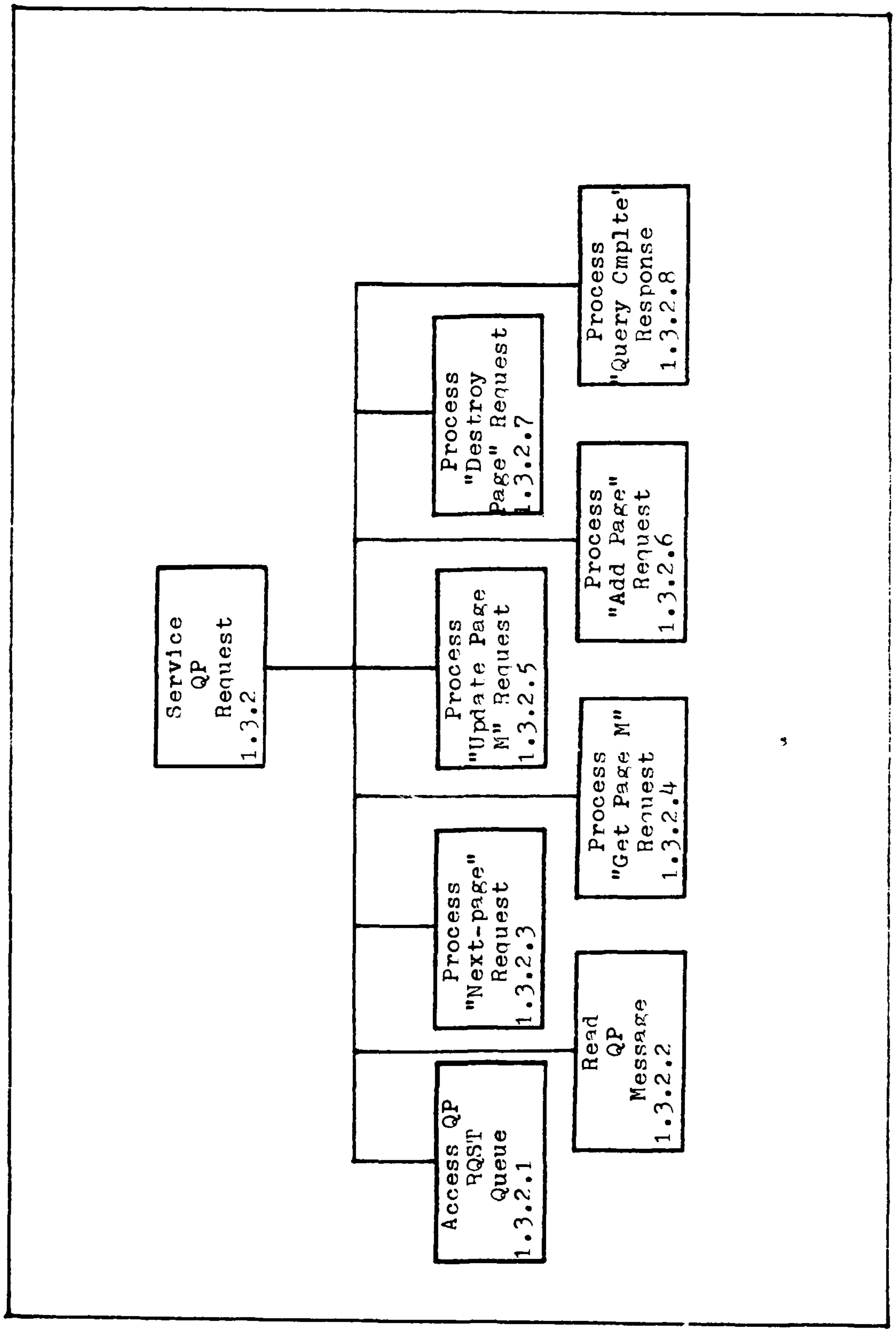
(Charts begin on following page)

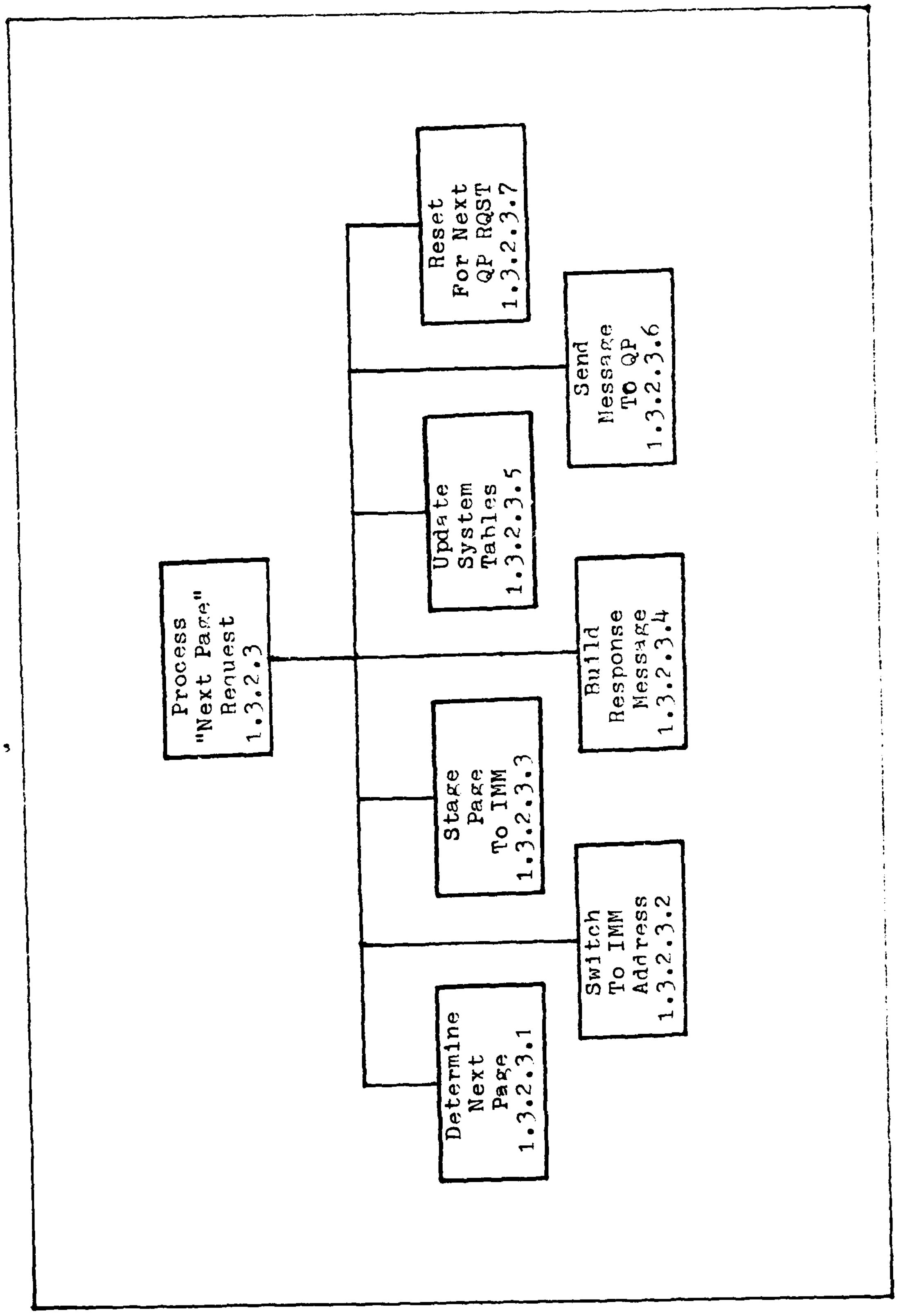

BCP
1.0
Initialize
1.1
Setup
1.2
Process
1.3
Wrapup
1.4


Initialize
1.1
Init
1.1.1

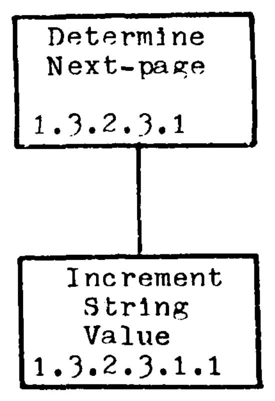

Setup
1.2
Init
1.1.1
Build
Domain Name
Table
1.2.1
Build
Attribute
Name Table
1.2.2
Build
Relation
Table
1.2.3
Build Rel's.
Attribute
Tables
1.2.4
Build Rel's.
Page
Tables
1.2.5
Build
QP
Table
1.2.6
Build
IMM
Tables
1.2.7
Setup
Configuration
1.2.8
Contact
Host
Computer
1.2.9

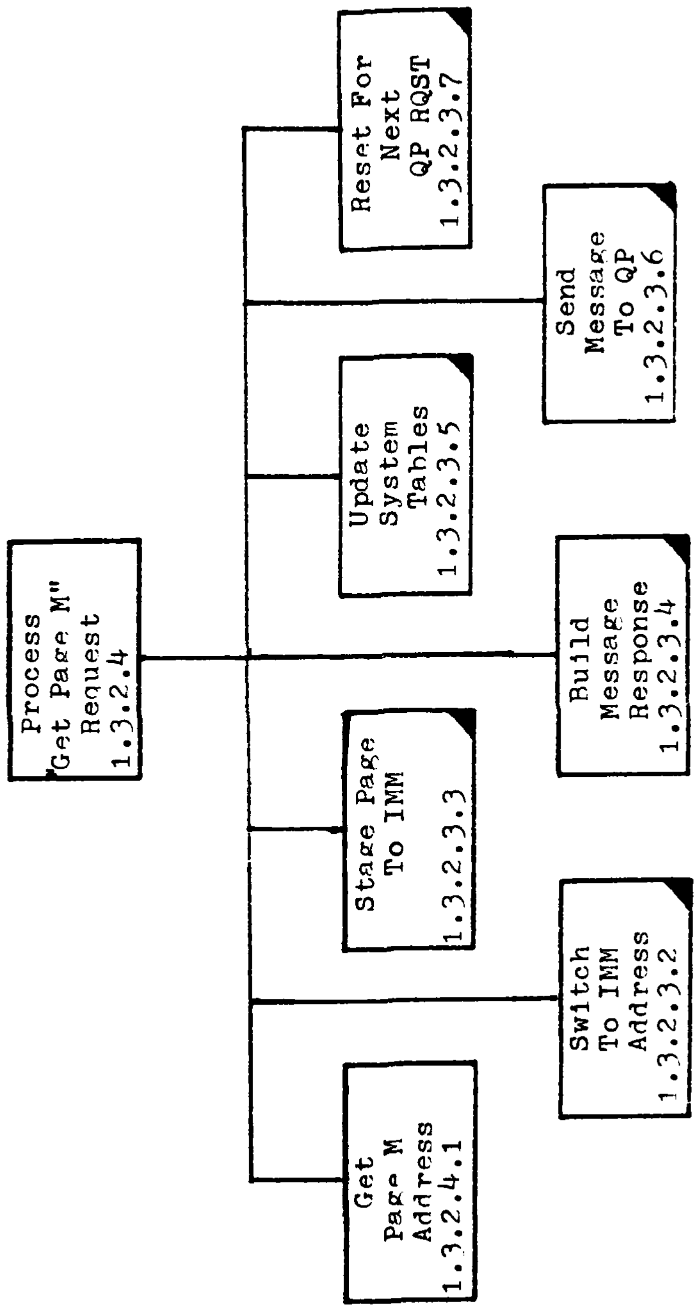

Process 1.3
Check Queues 1.3.1
Service QP Request 1.3.2
Service Host Request 1.3.3
Assign A QP 1.3.4
Transmit to Host 1.3.5
Backup Relational Database 1.3.6

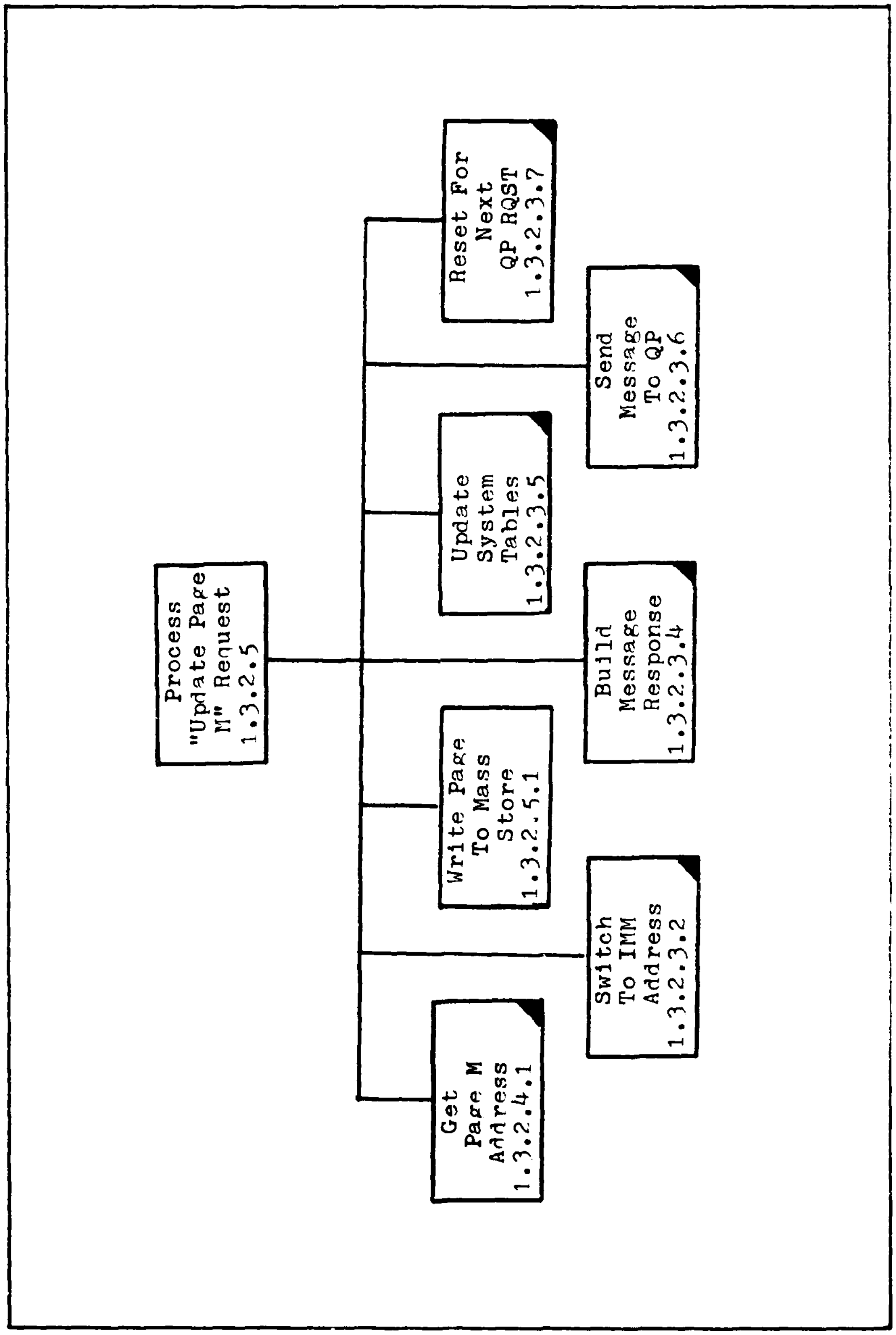

Service QP Request 1.3.2
Access QP RQST Queue 1.3.2.1
Read QP Message 1.3.2.2
Process "Next-page" Request 1.3.2.3
Process "Get Page M" Request 1.3.2.4
Process "Update Page M" Request 1.3.2.5
Process "Add Page" Request 1.3.2.6
Process "Destroy Page" Request 1.3.2.7
Process "Query Cmplte" Response 1.3.2.8

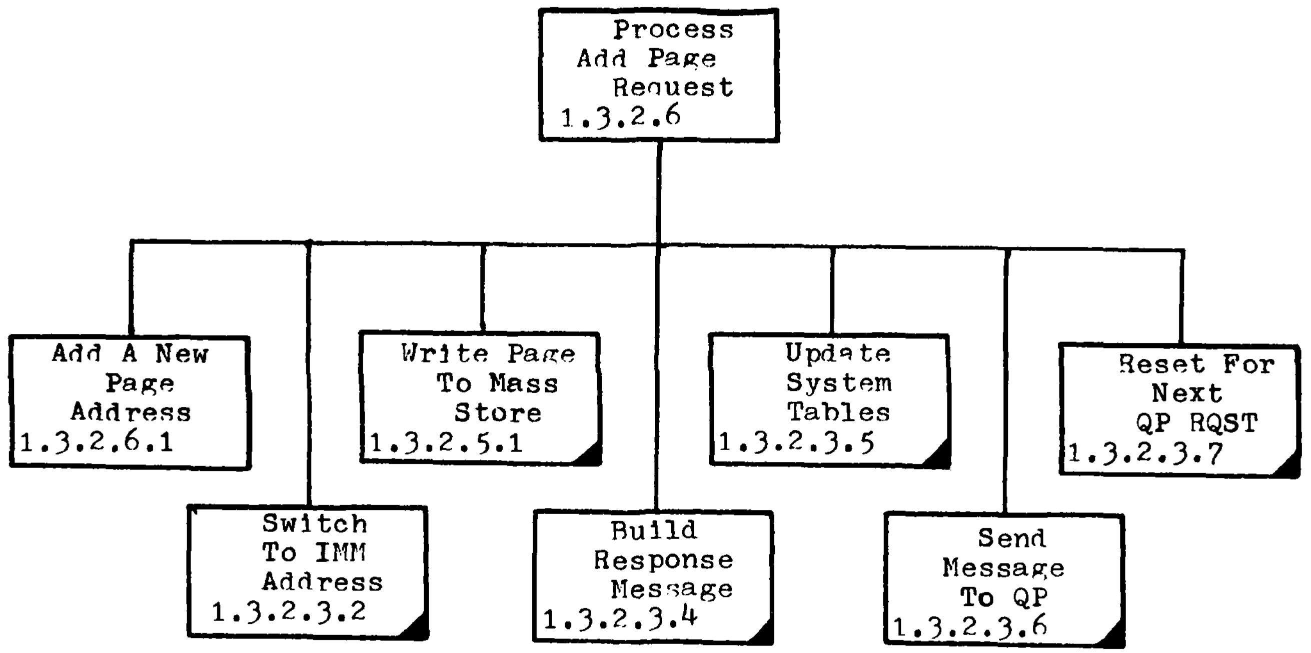


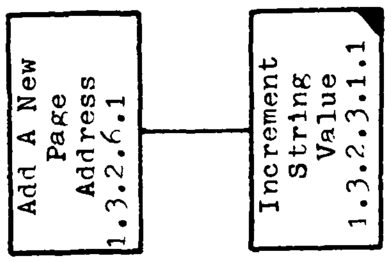
Process "Next Page" Request 1.3.2.3
Determine Next Page 1.3.2.3.1
Switch To IMM Address 1.3.2.3.2
Stage Page To IMM 1.3.2.3.3
Build Response Message 1.3.2.3.4
Update System Tables 1.3.2.3.5
Send Message TO QP 1.3.2.3.6
Reset For Next QP RQST 1.3.2.3.7

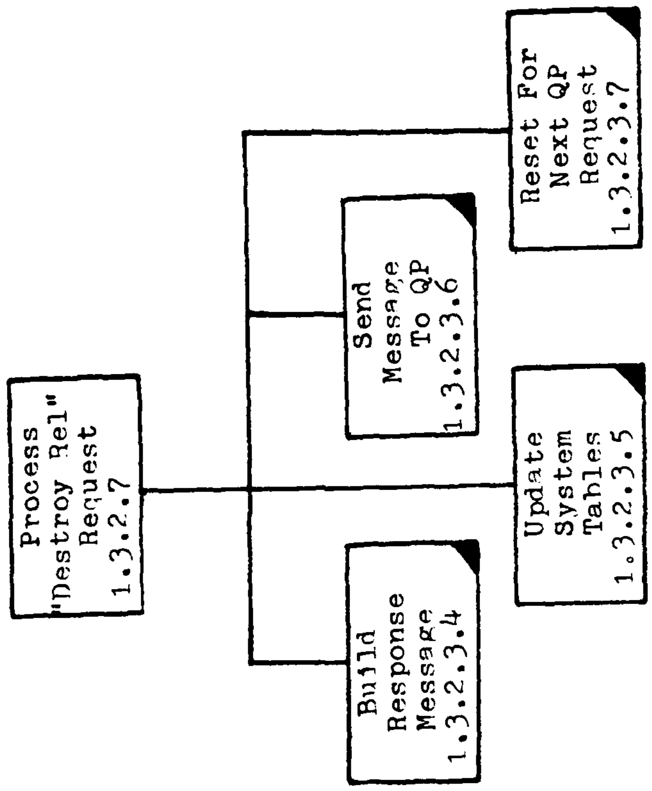

Determine
Next-page
1.3.2.3.1
Increment
String
Value
1.3.2.3.1.1

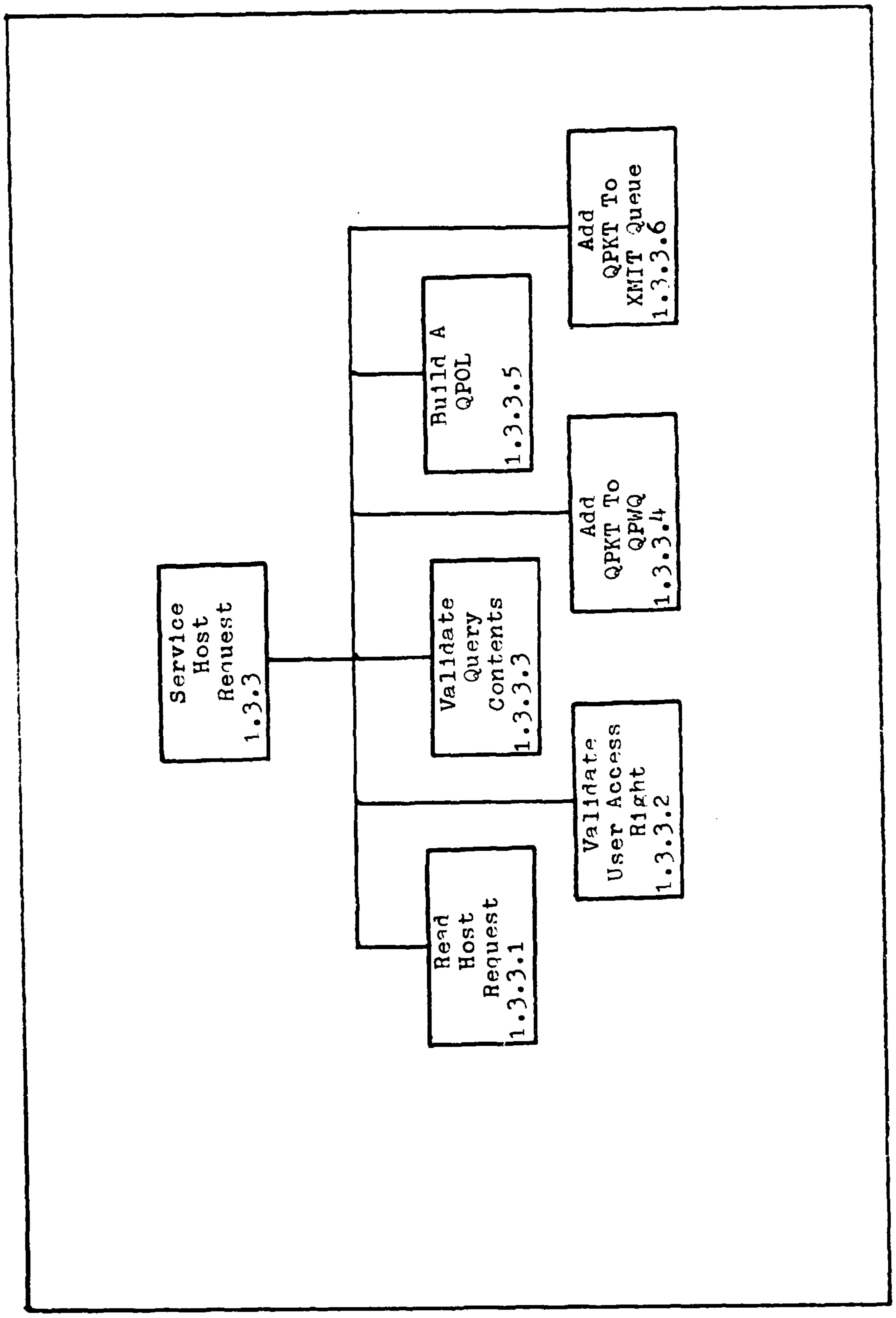

Process "Get Page M" Request 1.3.2.4
Get Page M Address 1.3.2.4.1
Switch To IMM Address 1.3.2.3.2
Stage Page To IMM 1.3.2.3.3
Build Message Response 1.3.2.3.4
Update System Tables 1.3.2.3.5
Send Message To QP 1.3.2.3.6
Reset For Next QP RQST 1.3.2.3.7

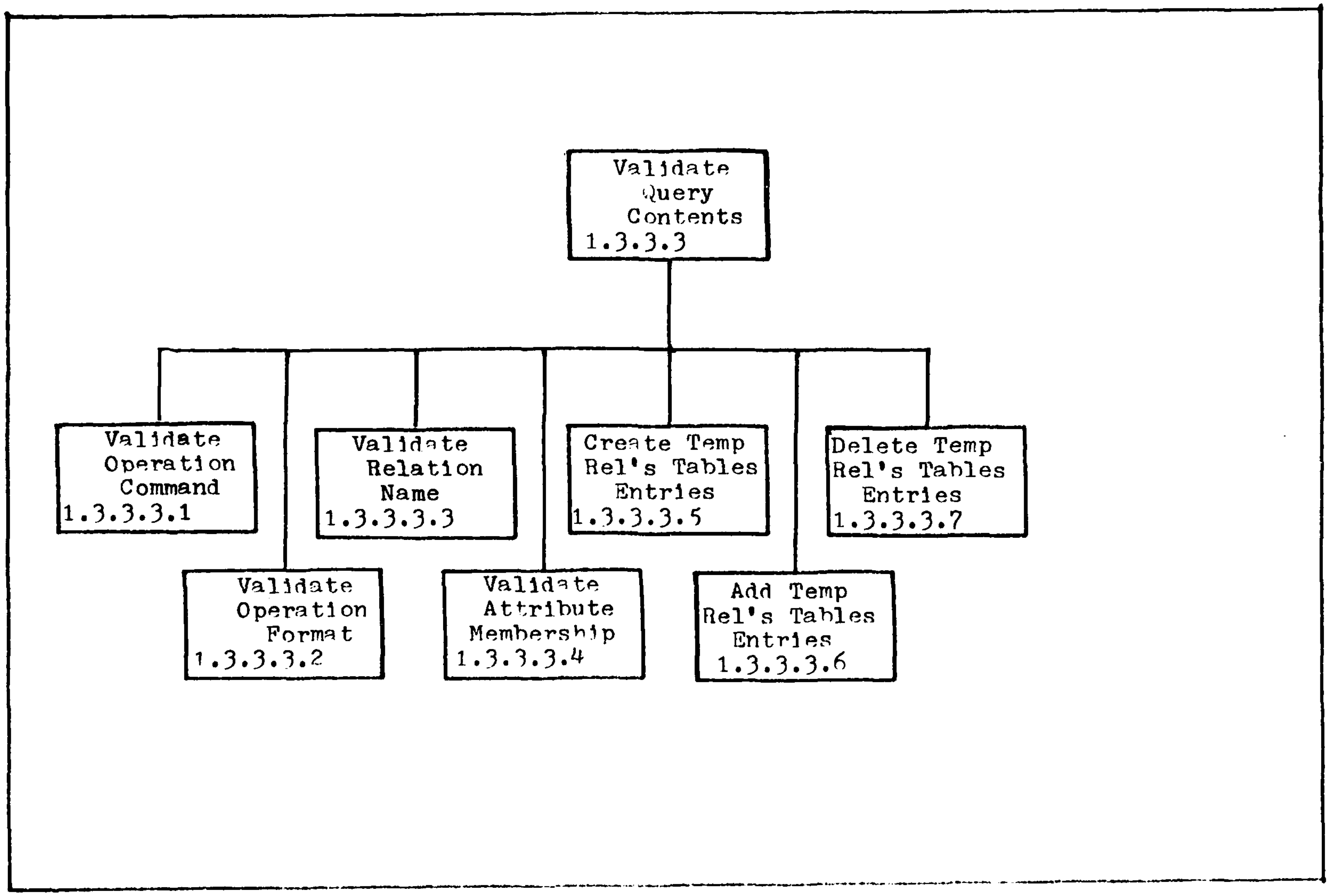

Process "Update Page M" Request 1.3.2.5
Get Page M Address 1.3.2.4.1
Switch To IMM Address 1.3.2.3.2
Write Page To Mass Store 1.3.2.5.1
Build Message Response 1.3.2.3.4
Update System Tables 1.3.2.3.5
Send Message To QP 1.3.2.3.6
Reset For Next QP RQST 1.3.2.3.7

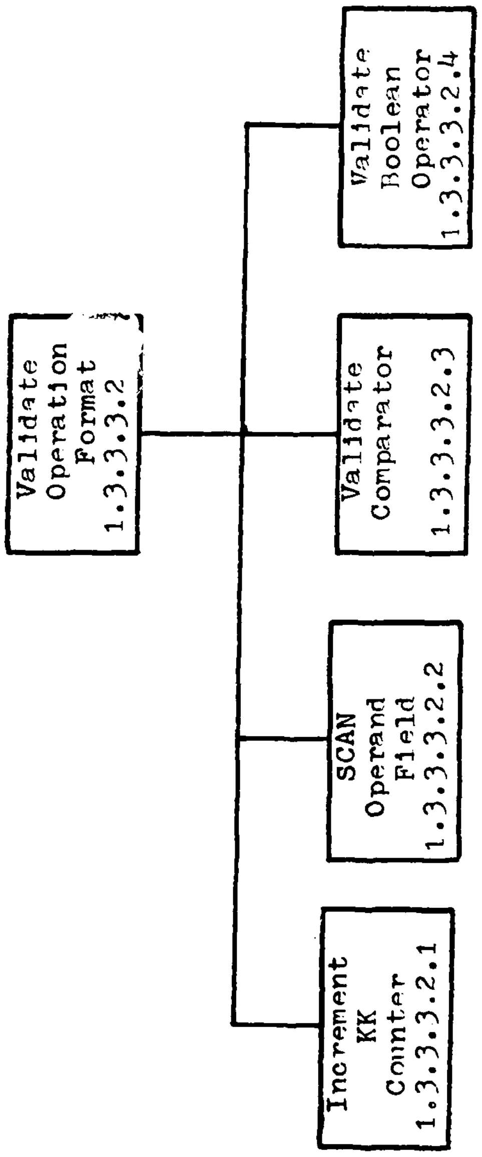

Process Add Page Request 1.3.2.6
Add A New Page Address 1.3.2.6.1
Switch To IMM Address 1.3.2.3.2
Write Page To Mass Store 1.3.2.5.1
Build Response Message 1.3.2.3.4
Update System Tables 1.3.2.3.5
Send Message To QP 1.3.2.3.6
Reset For Next QP RQST 1.3.2.3.7

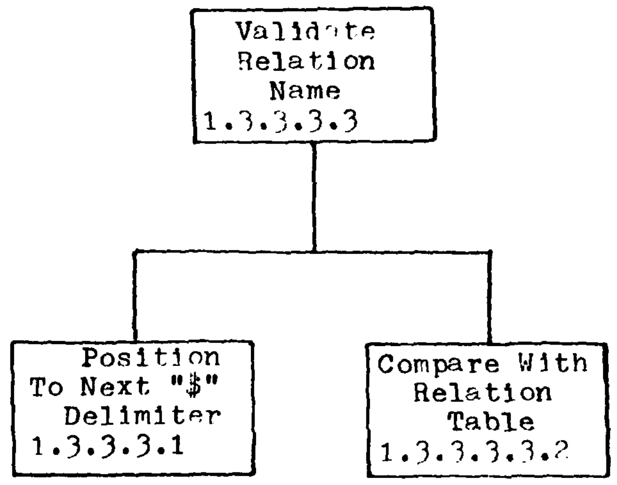

Add A New Page Address 1.3.2.6.1
Increment String Value 1.3.2.3.1.1

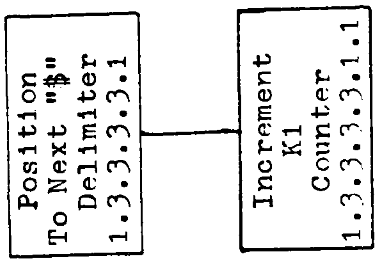

Process "Destroy Rel" Request 1.3.2.7
Build Response Message 1.3.2.3.4
Update System Tables 1.3.2.3.5
Send Message To QP 1.3.2.3.6
Reset For Next QP Request 1.3.2.3.7

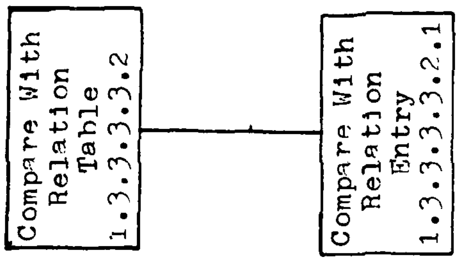

Service Host Request 1.3.3
Read Host Request 1.3.3.1
Validate User Access Right 1.3.3.2
Validate Query Contents 1.3.3.3
Add QPKT To QPWQ 1.3.3.4
Build A QPOL 1.3.3.5
Add QPKT To XMIT Queue 1.3.3.6

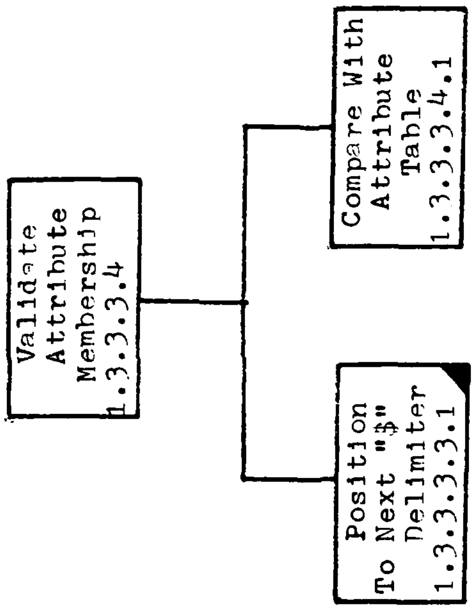

Validate Query Contents 1.3.3.3
Validate Operation Command 1.3.3.3.1
Validate Operation Format 1.3.3.3.2
Validate Relation Name 1.3.3.3.3
Validate Attribute Membership 1.3.3.3.4
Create Temp Rel's Tables Entries 1.3.3.3.5
Add Temp Rel's Tables Entries 1.3.3.3.6
Delete Temp Rel's Tables Entries 1.3.3.3.7

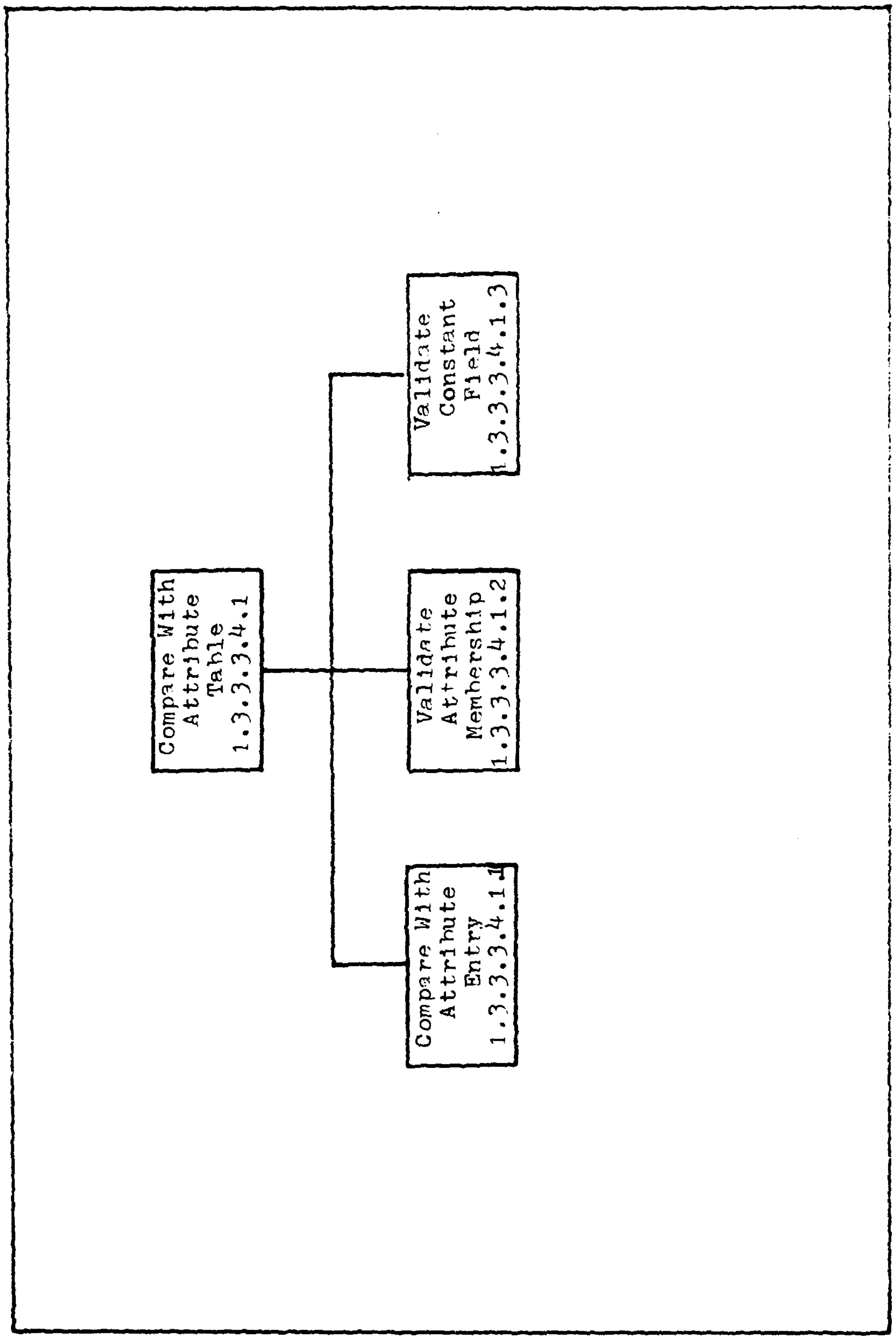

Validate Operation Format 1.3.3.3.2
Increment KK Counter 1.3.3.3.2.1
SCAN Operand Field 1.3.3.3.2.2
Validate Comparator 1.3.3.3.2.3
Validate Boolean Operator 1.3.3.3.2.4

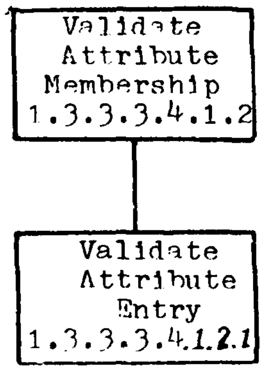

Validate Relation Name 1.3.3.3.3
Position To Next "$" Delimiter 1.3.3.3.1
Compare With Relation Table 1.3.3.3.3.2

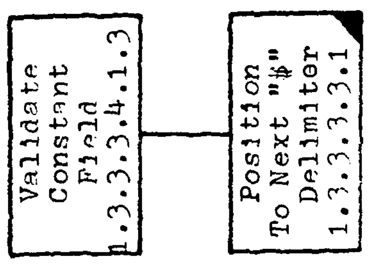

Position
To Next "$"
Delimiter
1.3.3.3.3.1
Increment
K1
Counter
1.3.3.3.3.1.1

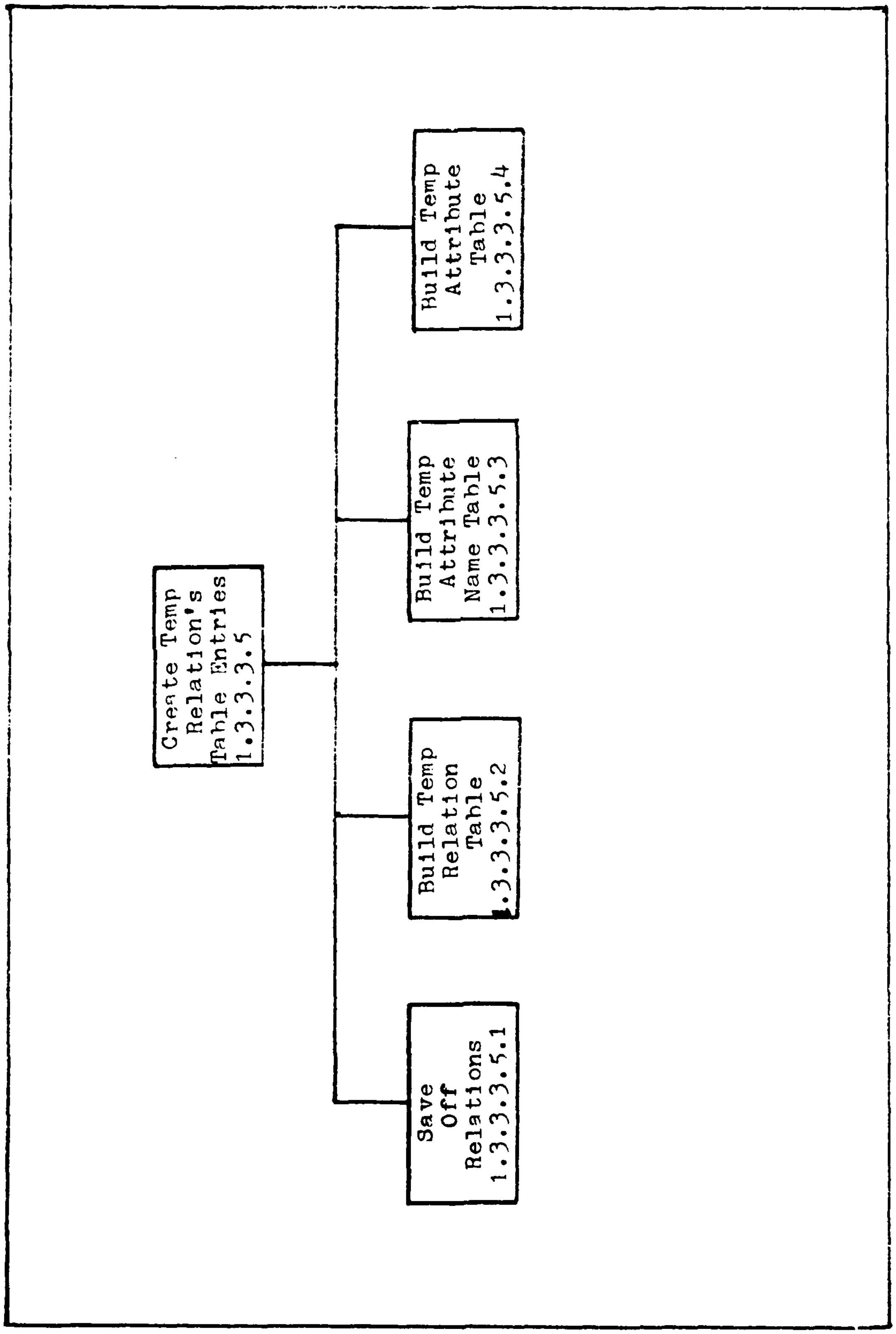

Compare With Relation Table 1.3.3.3.3.2
Compare With Relation Entry 1.3.3.3.3.2.1

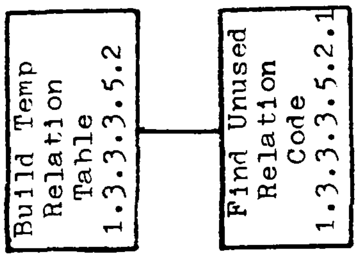

Validate Attribute Membership 1.3.3.3.4
Position To Next "$" Delimiter 1.3.3.3.3.1
Compare With Attribute Table 1.3.3.3.4.1

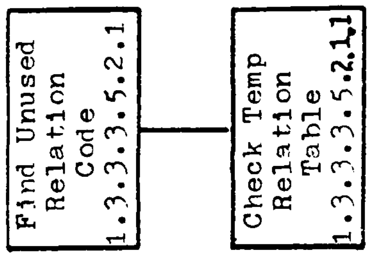

Compare With Attribute Table 1.3.3.3.4.1
Compare With Attribute Entry 1.3.3.3.4.1.1
Validate Attribute Membership 1.3.3.3.4.1.2
Validate Constant Field 1.3.3.3.4.1.3

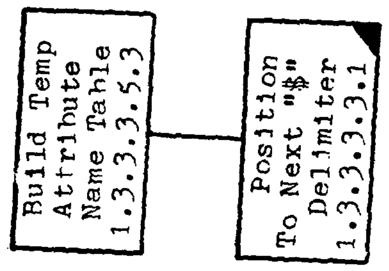

Validate Attribute Membership 1.3.3.3.4.1.2
Validate Attribute Entry 1.3.3.3.4.1.2.1

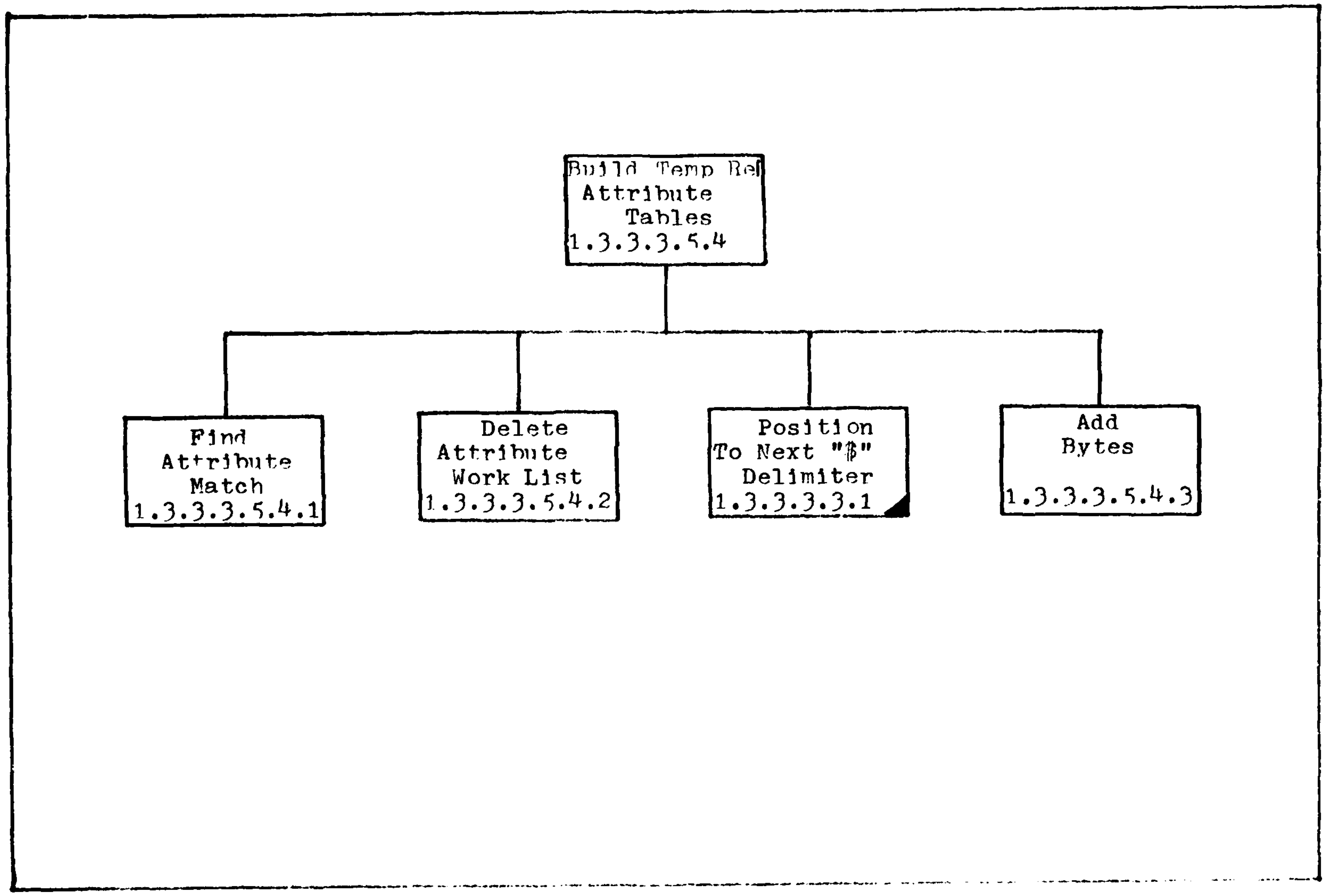

Validate Constant Field 1.3.3.3.4.1.3
Position To Next "$" Delimiter 1.3.3.3.3.1


Create Temp Relation's Table Entries 1.3.3.3.5
Save Off Relations 1.3.3.3.5.1
Build Temp Relation Table 1.3.3.3.5.2
Build Temp Attribute Name Table 1.3.3.3.5.3
Build Temp Attribute Table 1.3.3.3.5.4

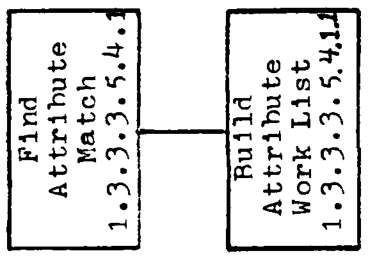

Build Temp Relation Table 1.3.3.3.5.2
Find Unused Relation Code 1.3.3.3.5.2.1

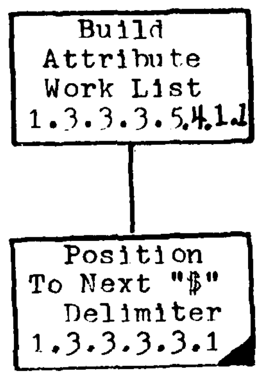

Find Unused Relation Code 1.3.3.3.5.2.1
Check Temp Relation Table 1.3.3.3.5.2.1.1

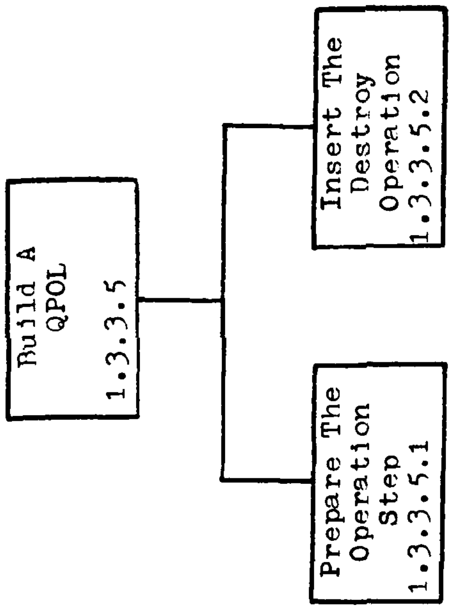

Build Temp Attribute Name Table 1.3.3.3.5.3
Position To Next "$" Delimiter 1.3.3.3.3.1

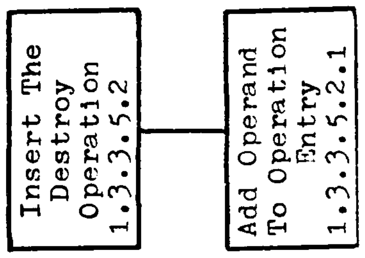


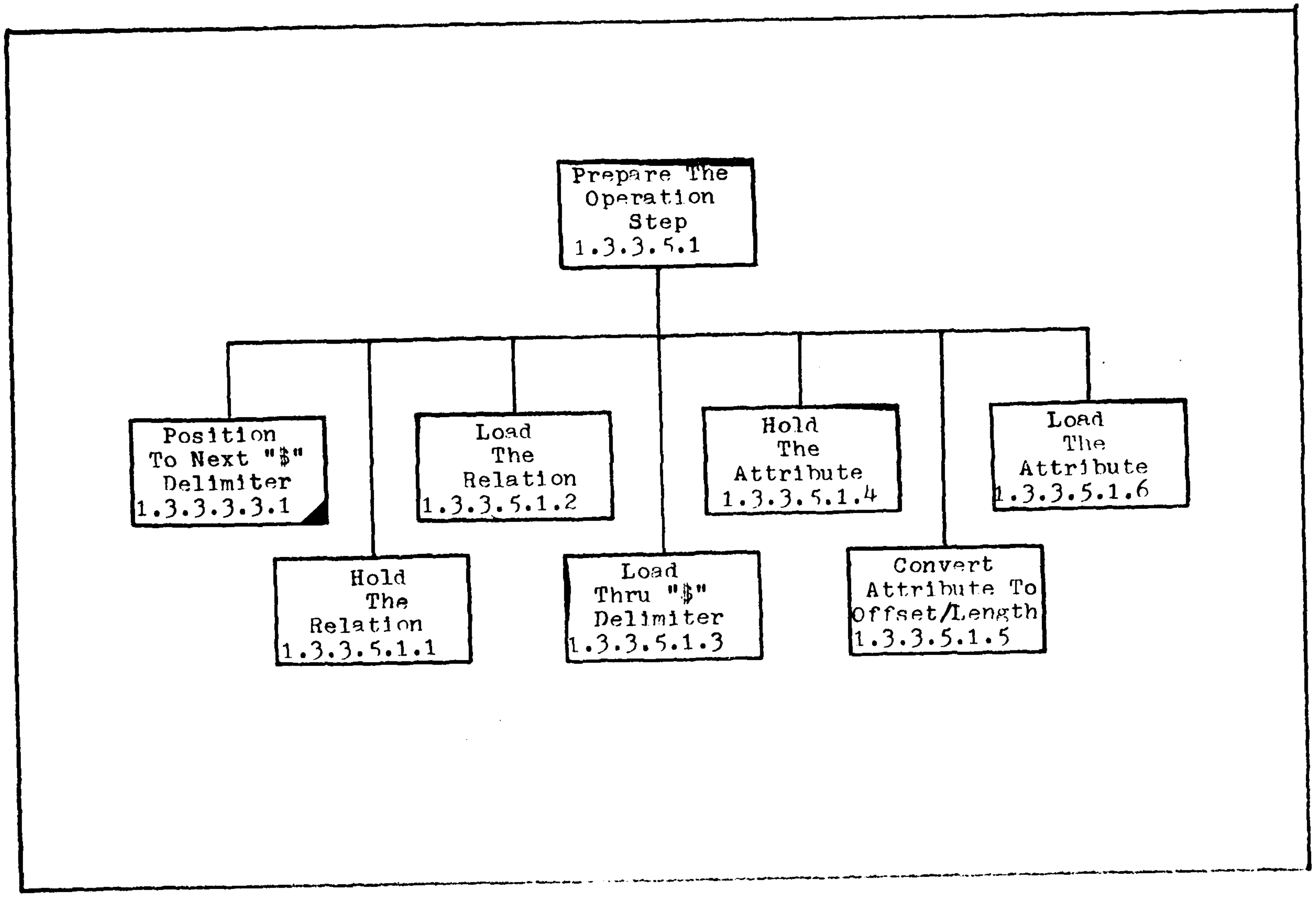
Build Temp Rel Attribute Tables 1.3.3.3.5.4
Find Attribute Match 1.3.3.3.5.4.1
Delete Attribute Work List 1.3.3.3.5.4.2
Position To Next "$" Delimiter 1.3.3.3.3.1
Add Bytes 1.3.3.3.5.4.3

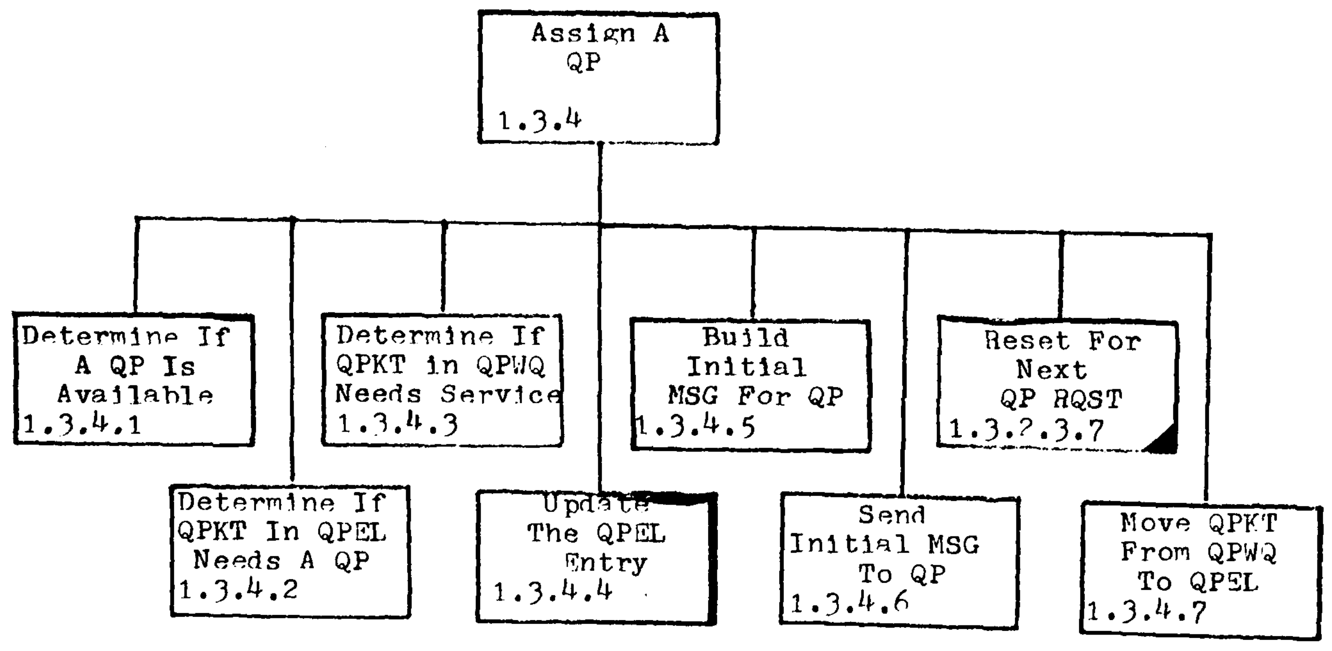

Find Attribute Match 1.3.3.3.5.4.1
Build Attribute Work List 1.3.3.3.5.4.1.1

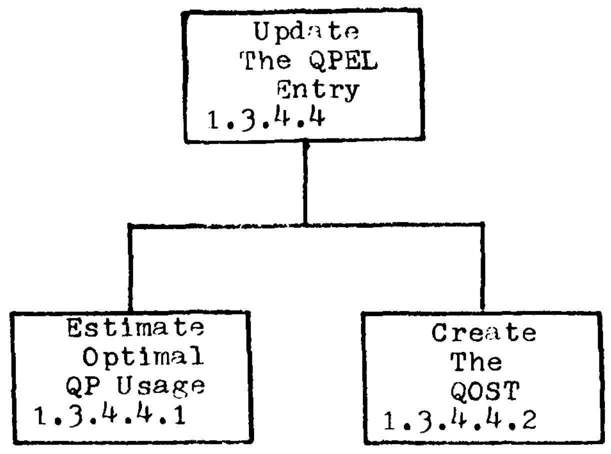

Build Attribute Work List 1.3.3.3.5.4.1.1
Position To Next "$" Delimiter 1.3.3.3.3.1


Build A QPOL 1.3.3.5
Prepare The Operation Step 1.3.3.5.1
Insert The Destroy Operation 1.3.3.5.2


Insert The Destroy Operation 1.3.3.5.2
Add Operand To Operation Entry 1.3.3.5.2.1


Prepare The Operation Step 1.3.3.5.1
Position To Next "$" Delimiter 1.3.3.3.3.1
Hold The Relation 1.3.3.5.1.1
Load The Relation 1.3.3.5.1.2
Load Thru "$" Delimiter 1.3.3.5.1.3
Hold The Attribute 1.3.3.5.1.4
Convert Attribute To Offset/Length 1.3.3.5.1.5
Load The Attribute 1.3.3.5.1.6

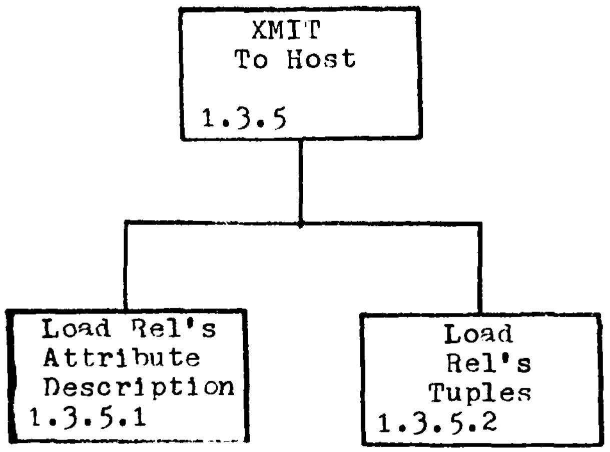

Assign A QP 1.3.4
Determine If A QP Is Available 1.3.4.1
Determine If QPKT In QPEL Needs A QP 1.3.4.2
Determine If QPKT in QPWQ Needs Service 1.3.4.3
Update The QPEL Entry 1.3.4.4
Build Initial MSG For QP 1.3.4.5
Send Initial MSG To QP 1.3.4.6
Reset For Next QP RQST 1.3.2.3.7
Move QPKT From QPWQ To QPEL 1.3.4.7

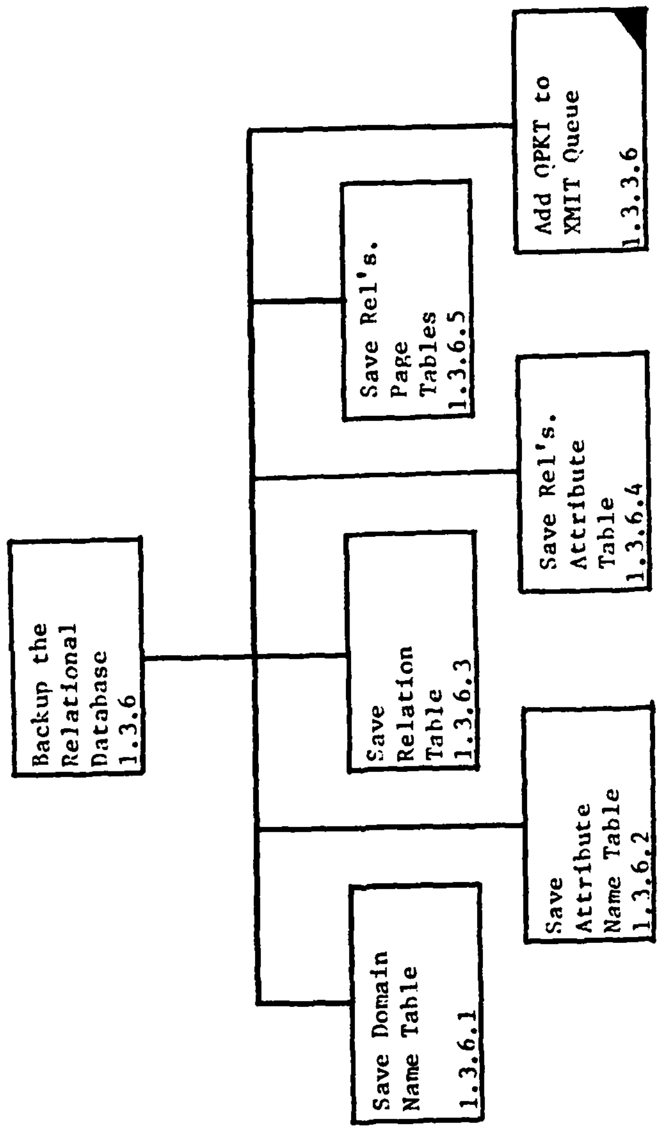

Update The QPEL Entry 1.3.4.4
Estimate Optimal QP Usage 1.3.4.4.1
Create The QOST 1.3.4.4.2

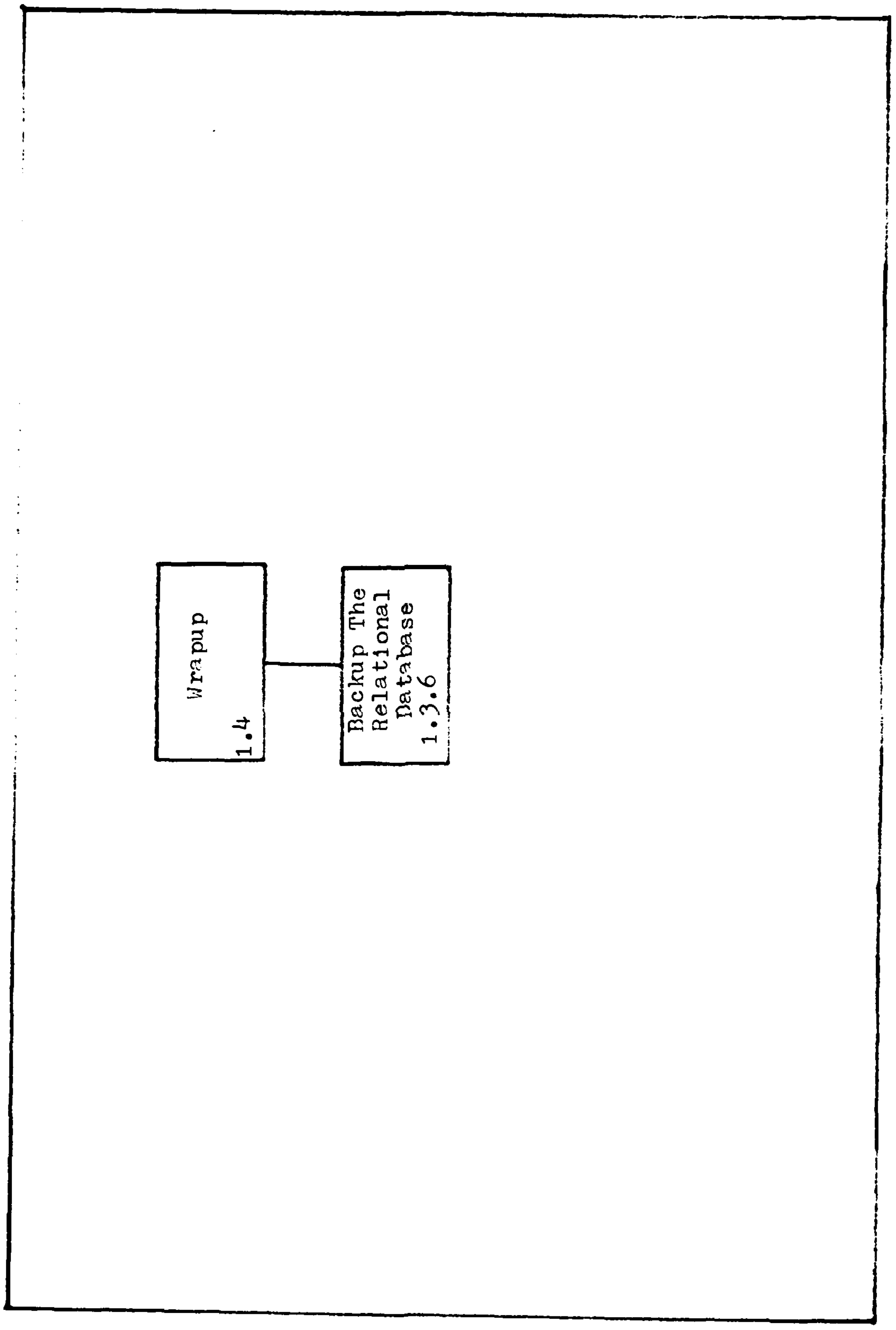
AD-A115 558 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
DESIGN AND IMPLEMENTATION OF A BACKEND MULTIPLE-PROCESSOR RELAT--ETC(U)
DEC 81 R W FONDEN
UNCLASSIFIED AFIT/GCS/EE/81D-6 NL

3 OF 3
AD
A 5559

END
DATE
FILMED
7 82
DTIC


XMIT
To Host
1.3.5
Load Rel's
Attribute
Description
1.3.5.1
Load
Rel's
Tuples
1.3.5.2


Backup the Relational Database 1.3.6
Save Domain Name Table 1.3.6.1
Save Attribute Name Table 1.3.6.2
Save Relation Table 1.3.6.3
Save Rel's. Attribute Table 1.3.6.4
Save Rel's. Page Tables 1.3.6.5
Add QPKT to XMIT Queue 1.3.3.6



Wrapup
1.4
Backup The Relational Database
1.3.6

Appendix B

Relational Algebra Query Operation Formats

Each relational algebra operation is described through the use of three columns of data: a field state, a maximum byte size length for the field, and a description of the field content.

In regards to the field state; R = required, and O = optional. Optional fields grouped together indicate an optional set which may exist in multiple set occurances.

A '$' indicates the termination of a data field (fields are variable length with a maximum restriction), a '%' indicates the end of the occurance of optional field sets, and a '#' indicates the end of a query operation. A '!' is used to indicate the end of the entire query request.

1) The SELECT operation format.

field state	max byte size	field description
R	6	"SELECT" command
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	attribute name
R	1	"$" delimiter
R	1	comparator
R	1	"$" delimiter
R	20	constant
R	1	"$" delimiter
O	3	boolean operator
O	1	"$" delimiter
O	20	attribute name
O	1	"$" delimiter
O	1	comparator
O	1	"$" delimiter
O	20	constant
O	1	"$" delimiter
R	1	"%" delimiter
R	20	relation name
R	1	"$" delimiter
R	1	"#" delimiter

2) The PROJECT operation format.

field state	max byte size	field description
R	7	"PROJECT" command
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	attribute name
R	1	"$" delimiter
O	20	attribute name
O	1	"$" delimiter
R	1	"%" delimiter
R	20	relation name
R	1	"$" delimiter
R	1	"#" delimiter

3) The JOIN operation format.

field state	max byte size	field description
R	4	"JOIN" command
R	1	"$" delimiter
R	20	relation name 1
R	1	"$" delimiter
R	20	relation name 2
R	1	"$" delimiter
R	20	attribute name 1
R	1	"$" delimiter
R	20	attribute name 2
R	1	"$" delimiter
O	20	attribute name 1
O	1	"$" delimiter
O	20	attribute name 2
O	1	"$" delimiter
R	1	"%" delimiter
R	20	relation name
R	1	"$" delimiter
R	1	"#" delimiter

4) The ADD (insert) operation format.

field state	max byte size	field description
R	3	"ADD" command
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	attribute value
R	1	"$" delimiter
O	20	attribute value
O	1	"$" delimiter
R	1	"%" delimiter
R	1	"#" delimiter

5) The DELETE operation format.

field state	max byte size	field description
R	6	"DELETE" command
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	attribute name
R	1	"$" delimiter
R	1	comparator
R	1	"$" delimiter
R	20	constant
R	1	"$" delimiter
O	3	boolean operator
O	1	"$" delimiter
O	20	attribute name
O	1	"$" delimiter
O	1	comparator
O	1	"$" delimiter
O	20	constant
O	1	"$" delimiter
R	1	"%" delimiter
R	1	"#" delimiter

6) The MODIFY operation format.

field state	max byte size	field description
R	6	"MODIFY" command
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	attribute name
R	1	"$" delimiter
R	1	comparator
R	1	"$" delimiter
R	20	constant
R	1	"$" delimiter
O	3	boolean operator
O	1	"$" delimiter
O	20	attribute name
O	1	"$" delimiter
O	1	comparator
O	1	"$" delimiter
O	20	constant
O	1	"$" delimiter
R	1	"%" delimiter
R	1	"#" delimiter

7) The COUNT operation format.

field state	max byte size	field description
R	6	"COUNT" command
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	attribute name
R	1	"$" delimiter
R	1	comparator
R	1	"$" delimiter
R	20	constant
R	1	"$" delimiter
O	3	boolean operator
O	1	"$" delimiter
O	20	attribute name
O	1	"$" delimiter
O	1	comparator
O	1	"$" delimiter
O	20	constant
O	1	"$" delimiter
R	1	"%" delimiter
R	1	"#" delimiter

8) The MAX, MIN, and AVE operations format.

field state	max byte size	field description
R	3	"MAX,MIN,AVE" cmd
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	attribute name
R	1	"$" delimiter
R	1	"#" delimiter

9) The CREATE operation format.

field state	max byte size	field description
R	6	"CREATE" command
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	attribute name
R	1	"$" delimiter
R	20	domain name
R	1	"$" delimiter
R	3	attribute length
R	1	"$" delimiter
R	1	attribute type
R	1	"$" delimiter
R	1	key member
R	1	"$" delimiter
O	20	attribute name
O	1	"$" delimiter
O	20	domain name
O	1	"$" delimiter
O	3	attribute length
O	1	"$" delimiter
O	1	attribute type
O	1	"$" delimiter
O	1	key member
O	1	"$" delimiter
R	1	"%" delimiter
R	1	"#" delimiter

10) The DESTROY and PRINT operations format.

field state	max byte size	field description
R	7	"DESTROY,PRINT" cmd
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	1	"#" delimiter

11) The UNION operation format.

field state	max byte size	field description
R	5	"UNION" command
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	20	relation name
R	1	"$" delimiter
R	1	"#" delimiter

Appendix C

BCP System Table Formats

The twelve BCP system tables, the four interprocessor communication message buffers, and all other work variables are defined in this appendix. This data is taken directly from the UCSD PASCAL computer program listing for the BCP subsystem software. All data entries are described by comments located to the right hand side of the listing.

```
(*$G+*)
(*$S*)
(*
                    AUTHOR : CAPT ROBERT W. FONDEN
                    CLASS  : GCS-81D
                    DATE   : 02 DECEMBER 1981
*)

UNIT COMMON;     (* SEGMENT NUMBER 10 *)

INTERFACE

CONST

     RELOPRSIZE = 197;               (* REL OPERATION SIZE     *)
     MSGSIZE = 1000;                 (* HOST MESSAGE SIZE      *)
     PAGESIZE = 4096;                (* PAGE SIZE              *)

     HOSTIN  =  07;                  (* HOST I UNITNUMBER VAL  *)
     HOSTOT  =  08;                  (* HOST O UNITNUMBER VAL  *)

     QP01IN  =  03;                  (* QP01 I UNITNUMBER VAL  *)
     QP02IN  =  06;                  (* QP02 I UNITNUMBER VAL  *)

     QP01OT  =  09;                  (* QP01 O UNITNUMBER VAL  *)
     QP02OT  =  10;                  (* QP02 O UNITNUMBER VAL  *)

     IM01IN  =  11;                  (* IMM01 I UNITNUMBER VAL *)
     IM01OT  =  12;                  (* IMM01 O UNITNUMBER VAL *)


TYPE

STG02 = STRING[02];
STG03 = STRING[03];
STG04 = STRING[04];
STG20 = STRING[20];
STG80 = STRING[80];

STRING02 = PACKED ARRAY  [1..02] OF CHAR;
STRING03 = PACKED ARRAY  [1..03] OF CHAR;
STRING04 = PACKED ARRAY  [1..04] OF CHAR;
STRING05 = PACKED ARRAY  [1..05] OF CHAR;
STRING06 = PACKED ARRAY  [1..06] OF CHAR;
STRING10 = PACKED ARRAY  [1..10] OF CHAR;
STRING14 = PACKED ARRAY  [1..14] OF CHAR;
STRING16 = PACKED ARRAY  [1..16] OF CHAR;
STRING20 = PACKED ARRAY  [1..20] OF CHAR;
STRING35 = PACKED ARRAY  [1..35] OF CHAR;
STRING55 = PACKED ARRAY  [1..55] OF CHAR;
STRING115  = PACKED ARRAY [1..115] OF CHAR;
STRINGOPER = PACKED ARRAY [1..RELOPRSIZE] OF CHAR;
STRINGMSSG = PACKED ARRAY [1..MSGSIZE] OF CHAR;
STRINGPAGE = PACKED ARRAY [1..PAGESIZE] OF CHAR;
```

```
DOMNAMEENTRY = RECORD                         (** DOMAIN NAME TABLE         **)
     NEXTDOMNAME : ^DOMNAMEENTRY;
     DOMNAME : STRING20;                      (** DOMAIN NAME               **)
     ATTMBRENTRY = RECORD                     (** DOMAIN'S ATTRIBUTE TABLE**)
          NEXTATTMBR : ^ATTMBRENTRY;
          DOMATTMBRPNTR : ^ATTNAMEENTRY; (** POINTER TO ATTNAMEENTRY **)
          END;
     END;

ATTNAMEENTRY = RECORD                         (* ATTRIBUTE NAME TABLE     *)
     NEXTATTNAME : ^ATTNAMEENTRY;
     ATTNAME : STRING20;                      (* ATTRIBUTE NAME            *)
     ATTLENGTH : STRING03;                    (* ATTRIBUTE BYTE LENGTH     *)
     ATTTYPER : CHAR;                         (* ATTRIBUTE TYPE            *)
     DOMPNTR : ^DOMNAMEENTRY;                (** POINTER TO DOMNAMEENTRY **)
     END;

RATTENTRY = RECORD                            (* RELATION ATTRIBUTE TAB   *)
     NEXTRATT : ^RATTENTRY;
     RATTNAME : STRING20;                     (* ATTRIBUTE NAME            *)
     BYTEOFF : STRING03;                      (* BYTE OFFSET               *)
     RATTTYPE : CHAR;                         (* ATTRIBUTE TYPE            *)
     RATTLEN : STRING03;                      (* ATTRIBUTE BYTE LENGTH     *)
     KEYMBR : CHAR;                           (* KEY MEMBER INDICATOR      *)
     END;

PAGEENTRY = RECORD                            (* RELATION PAGE TABLE       *)
     NEXTPAGE : ^PAGEENTRY;
     PAGENBR : STRING03;                      (* RELATION PAGE NUMBER      *)
     PAGEADDR : STRING16;                     (* RELATION PAGE ADDRESS     *)
     END;

RELENTRY = RECORD                             (* DB RELATION TABLE         *)
     NEXTREL : ^RELENTRY;
     RELNAME : STRING20;                      (* RELATION NAME             *)
     RELCODE : STRING03;                      (* RELATION CODE             *)
     ATPNTR : ^RATTENTRY;                     (* ATTRIBUTE TABLE POINTER *)
     PTPNTR : ^PAGEENTRY;                     (* PAGE TABLE POINTER        *)
     LOCKIND : CHAR;                          (* RELATION LOCK INDICATOR *)
     LOCKOWNER : STRING05;                    (* LOCK OWNER -MSGNBM-       *)
     TUPLELEN : STRING03;                     (* BYTE LENGTH OF TUPLES     *)
     ATTNBR : STRING02;                       (* NUMBER OF ATTRIBUTES      *)
     END;

IMMENTRY = RECORD                             (* A QP'S IMM TABLE          *)
     NEXTIMM : ^IMMENTRY;
     IMMADDR : STRING03;                      (* PORT ADDRESS OF IMM       *)
     IRELNAME : STRING20;                     (* RELATION NAME PAGED       *)
     END;

QPENTRY = RECORD                              (* QP TABLE                  *)
     NEXTQP : ^QPENTRY;
```

```
    QPNMBR : STRING02;                          (* QP ID NUMBER               *)
    QPSTATE : CHAR;                             (* STATE OF QP                *)
    IMMPNTR : ^IMMENTRY;                        (* IMM TABLE POINTER          *)
    END;

QPKTOPERENTRY = RECORD                          (* QPKT OPERATION STEPS       *)
    NEXTQPKTOPER : ^QPKTOPERENTRY;
    OPCODE : STRING03;                          (* OPERATION CODE             *)
    OPERANDS : STRINGOPER;                      (* SET OF OPERANDS            *)
    END;

QPWQENTRY = RECORD                              (* QPKT WAIT QUEUE            *)
    NEXTQPWQ : ^QPWQENTRY;
    INEX : CHAR;                                (* IN EXECUTION INDICTR       *)
    QPKTMSGNBR : STRING05;                      (* QPKT MESSAGE NUMBER        *)
    QPKTHOSTID : STRING06;                      (* QPKT HOST IDENT            *)
    QPKTPGMID : STRING06;                       (* QPKT PROGRAM ID            *)
    QPKTCRTID : STRING06;                       (* QPKT CRT ID                *)
    QPKTPRI : CHAR;                             (* QPKT PRIORITY              *)
    QPKTOPERPTR : ^QPKTOPERENTRY;               (* QPKT OPERATIONS TAB PTR    *)
    END;

QOSTENTRY = RECORD                              (* QUERY OPER STATE TABLE     *)
    NEXTQOST : ^QOSTENTRY;
    QOSTRELNAME : STRING20;                     (* RELATION NAME              *)
    RELPNTR : ^RELENTRY;                        (* POINTER TO REL ENTRY       *)
    QOSTPAGECUR : STRING03;                     (* RELATION PAGE CURRENCY     *)
    QOSTCNT : INTEGER;                          (* COUNT TOTAL                *)
    QOSTTOT : INTEGER;                          (* TOTAL VALUE                *)
    QOSTMIN : INTEGER;                          (* MINIMUM VALUE              *)
    QOSTMAX : INTEGER;                          (* MAXIMUM VALUE              *)
    END;

QPELENTRY = RECORD                              (* QPKT EXECUTE LIST          *)
    NEXTQPEL : ^QPELENTRY;
    QPWQPNTR : ^QPWQENTRY;                      (* QPWQ ENTRY POINTER         *)
    QPELMSGNBR : STRING05;                      (* QPKT MESSAGE NUMBER        *)
    OPTALLOC : INTEGER;                         (* OPTIMUM ALLOC OF QP'S      *)
    CURALLOC : INTEGER;                         (* CURRENT ALLOC OF QP'S      *)
    QOSTPNTR : ^QOSTENTRY;                      (* QOST POINTER               *)
    CMPLTDIND : CHAR;                           (* QPKT PROC COMPLETED IND    *)
    FNLRELNAME : STRING20;                      (* RELATION HOLDING ANSWER    *)
    END;

XMITQENTRY = RECORD                             (* TRANSMIT QUEUE FOR BCP     *)
    NEXTXMITQ : ^XMITQENTRY;
    XMSGNBR : STRING05;                         (* MESSAGE NUMBER             *)
    XHOSTID : STRING06;                         (* HOST ID NUMBER             *)
    XPGMID : STRING06;                          (* PROGRAM ID NUMBER          *)
    XCRTID : STRING06;                          (* CRT ID NUMBER              *)
    XPRI : CHAR;                                (* PRIORITY OF MESSAGE        *)
    XFNLRELNAME : STRING20;                     (* RELATION ANSWER NAME       *)
    XCOUNT : INTEGER;                           (* PAGE TRANSFER COUNTER      *)
    END;
```

```
VAR

     BPDOMNAMTBL   : ^DOMNAMEENTRY;  (* BASE POINTER OF DOMNAMTBL     *)
     BPATTNAMTBL   : ^ATTNAMEENTRY;  (* BASE POINTER OF ATTNAMTBL     *)
     BPRELTBL      : ^RELENTRY;      (* BASE POINTER OF RELTBL        *)
     BPQPTBL       : ^QPENTRY;       (* BASE POINTER OF QPTBL         *)
     BPQPWQ        : ^QPWQENTRY;     (* BASE POINTER OF QPWQ          *)
     BPQPEL        : ^QPELENTRY;     (* BASE POINTER OF QPEL          *)
     BPXMITQ       : ^XMITQENTRY;    (* BASE POINTER OF XMITQ         *)

     TBPDOMNAMTBL  : ^DOMNAMEENTRY;  (* TBASE POINTER OF DOMNAMTBL    *)
     TBPATTNAMTBL  : ^ATTNAMEENTRY;  (* TBASE POINTER OF ATTNAMTBL    *)
     TBPRELTBL     : ^RELENTRY;      (* TBASE POINTER OF RELTBL       *)


     CP0DNT        : ^DOMNAMEENTRY;
     CP1DNT        : ^DOMNAMEENTRY;
     CP2DNT        : ^DOMNAMEENTRY;  (* WORK POINTERS FOR DOMNAMETBL *)

     CP0AME        : ^ATTMBRENTRY;
     CP1AME        : ^ATTMBRENTRY;
     CP2AME        : ^ATTMBRENTRY;   (* WORK POINTERS FOR ATTMBRTBL  *)

     CP0ANT        : ^ATTNAMEENTRY;
     CP1ANT        : ^ATTNAMEENTRY;
     CP2ANT        : ^ATTNAMEENTRY;  (* WORK POINTERS FOR ATTNAMTBL  *)

     CP0RAT        : ^RATTENTRY;
     CP1RAT        : ^RATTENTRY;
     CP2RAT        : ^RATTENTRY;     (* WORK POINTERS FOR RATTTBL    *)

     CP0PT         : ^PAGEENTRY;
     CP1PT         : ^PAGEENTRY;
     CP2PT         : ^PAGEENTRY;     (* WORK POINTERS FOR PAGTBL     *)

     CP0RT         : ^RELENTRY;
     CP1RT         : ^RELENTRY;
     CP2RT         : ^RELENTRY;      (* WORK POINTERS FOR RELTBL     *)
     CP3RT         : ^RELENTRY;
     CP4RT         : ^RELENTRY;

     CP0IT         : ^IMMENTRY;
     CP1IT         : ^IMMENTRY;
     CP2IT         : ^IMMENTRY;      (* WORK POINTERS FOR IMMTBL     *)

     CP0QT         : ^QPENTRY;
     CP1QT         : ^QPENTRY;
     CP2QT         : ^QPENTRY;       (* WORK POINTERS FOR QPTBL      *)

     CP0QPT        : ^QPKTOPERENTRY;
     CP1QPT        : ^QPKTOPERENTRY;
```

```
    CP2QPT        : ^QPKTOPERENTRY; (* WORK POINTERS FOR QPKTOPERLST*)

    CP0QPWQ       : ^QPWQENTRY;
    CP1QPWQ       : ^QPWQENTRY;
    CP2QPWQ       : ^QPWQENTRY;     (* WORK POINTERS FOR QPWQ        *)

    CP0QOST       : ^QOSTENTRY;
    CP1QOST       : ^QOSTENTRY;
    CP2QOST       : ^QOSTENTRY;     (* WORK POINTERS FOR QOST        *)

    CP0QPEL       : ^QPELENTRY;
    CP1QPEL       : ^QPELENTRY;
    CP2QPEL       : ^QPELENTRY;     (* WORK POINTERS FOR QPEL        *)

    CP0XMITQ      : ^XMITQENTRY;
    CP1XMITQ      : ^XMITQENTRY;
    CP2XMITQ      : ^XMITQENTRY;    (* WORK POINTERS FOR XMITQ       *)


(* FOR MESSAGES GOING FROM QP'S TO BCP                               *)

    QBMSGCODE : CHAR;        (* A C D G N U  A = ADD-PAGE REQUEST     *)
    QBQPNMBR : STRING02;     (* # # # # # #  C = PROC CMPLT RESPONSE *)
    QBMSGNBR : STRING05;     (* # # # # # #  D = DESTROY REL REQUEST *)
    QBRELNAME : STRING20;    (* #   # # # #  G = GET-PAGE REQUEST     *)
    QBIMMLOC : STRING03;     (* #     # # #  N = NEXT-PAGE REQUEST    *)
    QBPAGENBR : STRING03;    (*       #   #  U = UPDATE-PAGE REQUEST *)
    QBCOUNT : STRING20;      (*         #                             *)
    QBTOTAL : STRING20;      (*         #                             *)
    QBMIN : STRING20;        (*         #                             *)
    QBMAX : STRING20;        (*         #                             *)
    QBLOCK : CHAR;           (*       # #                             *)


    QBBUFFER   : STRING115;

    QB01BUFFER : STRING115;
    QB02BUFFER : STRING115;
    QB03BUFFER : STRING115;
    QB04BUFFER : STRING115;
    QB05BUFFER : STRING115;
    QB06BUFFER : STRING115;
    QB07BUFFER : STRING115;
    QB08BUFFER : STRING115;

(* FOR MESSAGES GOING FROM BCP TO QP'S. [ANSWERING QP MESSAGES ONLY} *)

    BQANSCODE : CHAR;     (* A D L N P U  A = PAGE HAS BEEN ADDED   *)
    BQRLNNAME : STRING20; (*         #    D = RELATION IS DESTROYED *)
    BQPAGENBR : STRING03; (*         #    L = REL LOCKED, TRY LATER *)
    BQIMMADDR : STRING03; (*         #    N = ARE NO MORE PAGES     *)
```

```
                         (*            P = HERE IS ANOTHER PAGE  *)
                         (*            U = PAGE WAS REWRITTEN    *)


     BQBUFFERA : STRING35;
     BQBUFERB  : STRINGMSSG;

(* FOR MESSAGES GOING BETWEEN THE BCP AND THE HOST                *)

     HBBUFFER : STRINGMSSG;   (* HOST TO BCP MESSAGE BUFFER       *)
     BHBUFFER : STRINGMSSG;   (* BCP TO HOST MESSAGE BUFFER       *)


(* FILE ID'S AND TITLE NAMES FOR THE FILES CONTAINING SYSTEM TABLES *)

     PAGEFILE : TEXT;           (* FILE-ID TO REFERENCE PAGE FILES  *)
     TBLFILE  : TEXT;           (* FILE-ID TO REFERENCE TABLE FILES *)

     ATTNAMTBL : STG20;         (* DBNN:ATTNAMTBL.TEXT              *)
     RELTBL : STG20;            (* DBNN:RELTBL.TEXT                 *)
     RATTTBLS : STG20;          (* DBNN:RATTTBLS.TEXT               *)
     PAGTBLS : STG20;           (* DBNN:PAGTBLS.TEXT                *)
     TRFILE : STG20;            (* DBNN:TR1FILE.TEXT                *)



(*   MISCALANEOUS STRING VARIABLES                                  *)

     CH : CHAR;                 (* FOR CONSOLE BYTE/STRING INPUT    *)
     BCPRESP : STG20;           (* FUNCTION ANSWER TO QUERY RQST    *)
     DOLLAR : CHAR;             (* A '$' DELIMITER                  *)
     HOLDATT : STRING20;        (* ATTRIBUTE NAME HOLD BUFFER       *)
     HOLDPAGE : STRING03;       (* RELATION PAGE HOLD BUFFER        *)
     HOLDREL : STRING20;        (* RELATION NAME HOLD BUFFER        *)
     HOSTMSGLEN : STRING05;     (* BYTE LENGTH OF NEXT HOST MESSAGE *)
     KEYBRD : STG80;            (* OPERATOR CONSOLE INPUT           *)
     NQPS : STG02;              (* NUMBER OF QP'S CONFIGURED FOR RUN *)
     PAGEBUFFER : STRINGPAGE;   (* PAGE STAGING BUFFER TO IMM       *)
     PAGETOSTAGE : STG20;       (* PAGE ADDRESS OF PAGE TO STAGE NEXT*)
     PAGETOWRITE : STG20;       (* PAGE ADDRESS OF PAGE TO WRITE BACK*)
     PGM : STRING55;            (* ALGORITHM FOR SERVICE SELECTION  *)
     QPVALUE : STRING16;        (* QP CONFIGURATION STARTUP TABLE   *)
     RELVOL1ID : STG04;         (* VOLUME 1 HOLDING RELATION FILES  *)
     RELVOL2ID : STG04;         (* VOLUME 2 HOLDING RELATION FILES  *)
     RSPNS : CHAR;              (* RESPONSE INDICATOR BETWEEN PGMS  *)
     VOLTBLS : STG04;           (* VOLUME # HOLDING DB SYSTEM TABLES *)
     VOPCODE : STRING03;        (* QUERY STEP OP-CODE TEMP-BUFFER   *)

(*   MISCALANEOUS INTEGER VARIABLES                                 *)
```

```
     CLOCK : INTEGER;              (* PGM COUNTER                         *)
     COUNT : INTEGER;              (* UTILITY COUNTER                     *)
     DBSAVE : INTEGER;             (* 1 = DB SAVE HAS BEEN REQUESTED      *)
     DISP : INTEGER;               (* BYTE DISPLACEMENT COUNT             *)
     ER : INTEGER;                 (* STRING-INTEGER;ERROR INDICATOR      *)
     FF : INTEGER;                 (* FIRST BYTE POSITION POINTER         *)
     FOUNDIT : INTEGER;            (* INDICATE IF FILE IS ON-LINE OR NOT*)
     HBYTEPOS : INTEGER;           (* INDEX USED FOR SCANNING HBBUFFER  *)
     HOLDIORSLT : INTEGER;         (* SAVE OFF THE IORESULT OF I/O CALL *)
     HOLDRQSTIND : INTEGER;        (* FOR DBSAVE / WRAPUP REQUESTS        *)
     I,J,K,L,M,N : INTEGER;        (* UTILITY COUNTERS                    *)
     II,JJ,KK,LL,MM : INTEGER;     (* UTILITY COUNTERS                    *)
     IMMNBR : INTEGER;             (* IMM TO RECIEVE STAGED REL PAGE      *)
     LAST : INTEGER;               (* LAST QP SERVICED                    *)
     MAX : INTEGER;                (* MAXIMUM ALLOWABLE STRING LENGTH     *)
     NOTT : INTEGER;               (* A STATUS INDICATOR                  *)
     NQPSVALUE : INTEGER;          (* NUMBER OF QP'S CONFIGURED TO PROC *)
     NVOLS : INTEGER;              (* NBR OF VOL'S HOLDING DB RELATIONS *)
     PROCIDLE : INTEGER;           (* PROC IDLE - FOR DBSAVE / WRAPUP     *)
     QPCHOSEN : INTEGER;           (* QP SELECTED TO BE SERVICED NEXT     *)
     QPWQFULL : INTEGER;           (* 1 = QPWQ IS PRESENTLY FULL          *)
     RELRQSTIND : INTEGER;         (* FOR DBSAVE / WRAPUP RESPONSES       *)
     RESULT : INTEGER;             (* DECISION INDICATOR                  *)
     RESPONSE : INTEGER;           (* DECISION INDICATOR                  *)
     RR : INTEGER;                 (* LAST BYTE (REAR) BYTE POINTER       *)
     WRAPUP : INTEGER;             (* 1 = WRAPUP HAS BEEN REQUESTED       *)


(*-------------------------------------------------------------------*)

END. (* UNIT COMMON [ SEGMENT NUMBER 10 ] *)
```

Appendix D

BCP System Data Base File Formats

The following six files are used to support this system.

1) domain name file
2) attribute name file
3) relations file
4) relation's attributes file
5) relation's pages file
6) relation's page files (pages of tuples)

The specific formats for each of these files are defined in the following paragraphs. These files all store their data in UCSD text format. Note that the first four files also contain carriage-return (<cr>) delimiters between each of the records. This is to enable both easy inspection of these files using the UCSD PASCAL editor, as well as easy modification, using the UCSD PASCAL editor, for testing purposes.

(1) Domain Name File Record Format.

01 - 20	domain name
21 - 40	attribute name

Example File:

```
[BOF]
NAME          STUDENTNAME        <CR>
NAME          FACULTYNAME        <CR>
SSAN          STUDENTSSAN        <CR>
SSAN          FACULTYSSAN        <CR>
SSAN          EDPLANSSAN         <CR>
GRADE         EDPLANGRADE        <CR>
COURSE        EDPLANCOURSE       <CR>
QUARTER       EDPLANQUARTER      <CR>
PROGRAM       STUDENTPROGRAM     <CR>
EDCODE        STUDENTEDCODE      <CR>
DEPT          FACULTYDEPT        <CR>
[EOF]
```

(2) Attribute Name File Record Format.

01 - 20	attribute name
21 - 23	byte length
24	attribute type
25 - 44	domain name
45	<cr>

Example File:

```
[BOF]
STUDENTNAME       020CNAME          <CR>
STUDENTSSAN       009CSSAN          <CR>
STUDENTPROGRAM    006CPROGRAM       <CR>
```

```
STUDENTEDCODE       004CEDCODE              <CR>
FACULTYNAME         009CNAME                <CR>
FACULTYSSAN         009CSSAN                <CR>
FACULTYDEPT         003CDEPT                <CR>
EDPLANSSAN          009CSSAN                <CR>
EDPLANCOURSE        005CCOURSE              <CR>
EDPLANQUARTER       004CQUARTER             <CR>
EDPLANGRADE         002CGRADE               <CR>
[EOF]
```

(3) Relations File Record Format.

01 - 20	relation name
21 - 23	relation code
24 - 26	tuple byte length
27 - 28	number of attributes
29	<CR>

Example File:

```
[BOF]
STUDENT             00103904<CR>
FACULTY             00203203<CR>
EDPLAN              00302004<CR>
[EOF]
```

(4) Relation's Attribute File Record Format.

01 - 20	attribute name
21 - 23	byte offset in tuple
24	attribute type
25 - 27	byte length of attribute
28	key membership indicator
29	<CR>

Example File:

```
[BOF]
STUDENTNAME          000C020N<CR>
STUDENTSSAN          021C009Y<CR>
STUDENTPROGRAM       030C006N<CR>
STUDENTEDCODE        036C004N<CR>
$<CR>
FACULTYNAME          000C020N<CR>
FACULTYSSAN          021C009Y<CR>
FACULTYDEPT          030C003N<CR>
$<CR>
EDPLANSSAN           000C009Y<CR>
EDPLANCOURSE         009C005N<CR>
EDPLANQUARTER        014C004N<CR>
EDPLANGRADE          018C002N<CR>
$<CR>
[EOF]
```

Note that a '$' delimiter is used to group a set of attributes defined for each relation. Each set of attributes are sequenced in accordance with the order of the relations stored in the relation file.

(5) Relation's Page File Record Format.

01 - 03	page number
04 - 07	volume-id
08	':'
09 - 19	page file address name
09 - 11	relation code
12 - 14	page number
15 - 19	'.text'
20	<CR>

Example File:

```
[BOF]
001DBV1:001001.TEXT<CR>
002DBV1:001002.TEXT<CR>
003DBV1:001003.TEXT<CR>
$<CR>
001DBV1:002001.TEXT<CR>
002DBV1:002002.TEXT<CR>
$<CR>
001DBV2:003001.TEXT<CR>
002DBV2:003002.TEXT<CR>
```

```
003DBV2:003003.TEXT<CR>
003DBV1:003004.TEXT<CR>
$<CR>
[EOF]
```

Note that a '$' delimiter is used to group a set of pages existing for each relation. Each set of pages are sequenced in accordance with the order of the relations stored in the relation file.

(6) Relation's Page Files Record Format.

0001	'/' BOP delimiter
0002 - NNNN	relation's tuples
NNNN + 1	'\' EOP delimiter
NNNN + 2 - MMMM	unused space in page

where MMMM = the fixed byte size of the pages.

Appendix E

Inter-processor Architectural Specifications

Figure 26 portrays the inter-processor architecture used to accomplish the testing for stage one of this development effort. Five processors were used to accomplish the testing for developing the BCP's software. The acronyms DLV-11, DLV-11J, DUALPORT, and DRV- 11J respectively identify a single port 38.4 Kbaud serial line channel card, a quad port 38.4 Kbaud serial line channel card, a dual port 38.4 Kbaud serial line channel with 16 Kbyte RAM card, and a single port parallel line channel card. Fifty six Kbytes of RAM are resident on each of the five systems. A CRT is assigned to each processor subsystem for supporting the subsystem development. The finalized system will only require a CRT attached to the BCP.

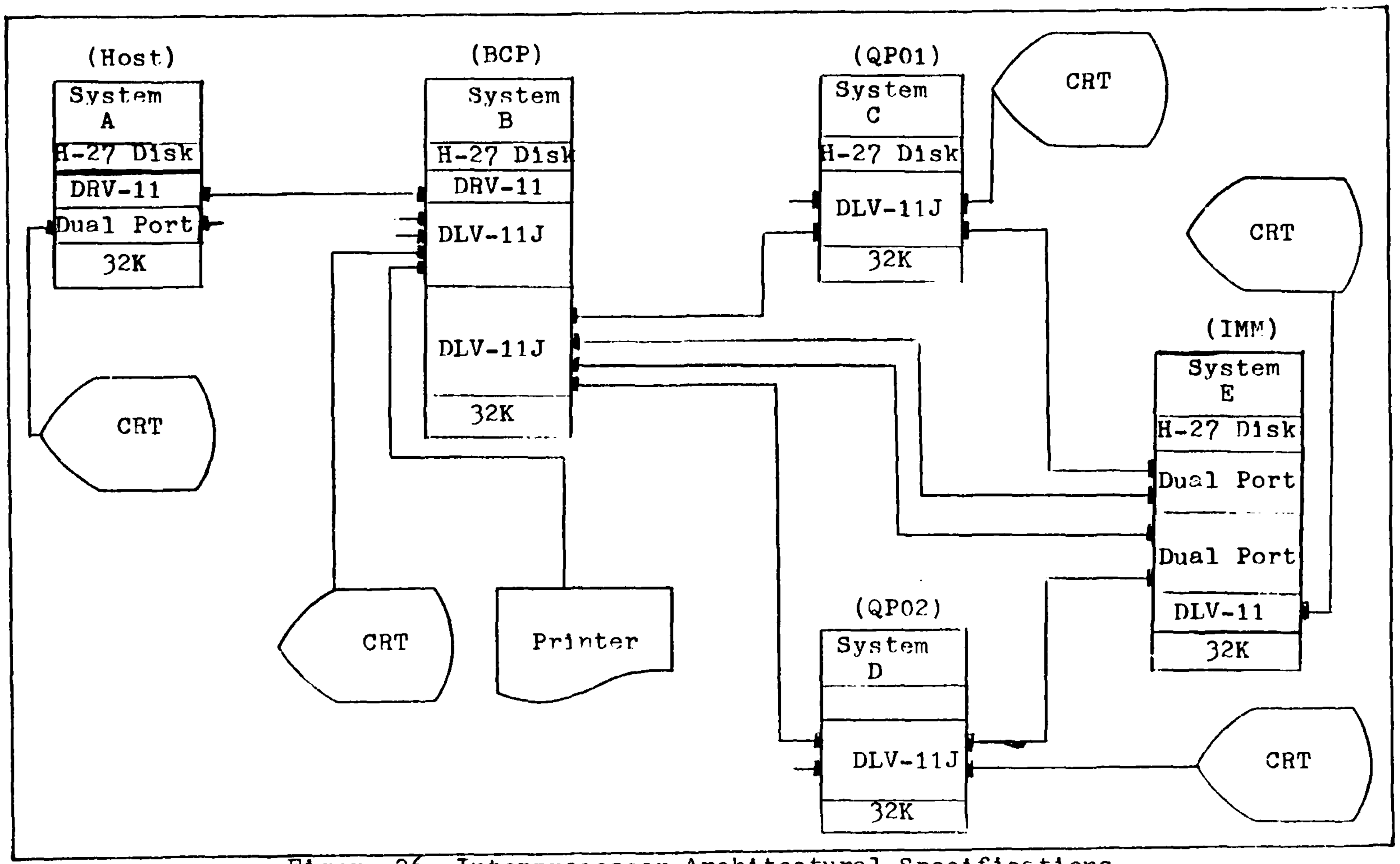


Figure 26 Interprocessor Architectural Specifications

Appendix F

Channel Link Software Specifications

This appendix contains all the necessary information to construct channel link communication software drivers and incorporate them into the UCSD PASCAL operating system software package known as SYSTEM.PDP11.

To add a new channel link communication software driver, several decisions must first be made. First, a UCSD PASCAL UNITNUMBER value must be selected and then a channel number must be selected for the UNITNUMBER to be assigned to. For example, UNITNUMBER 7 and channel 0 were selected for the first driver, and UNITNUMBER 8 and channel 2 were selected for the second driver. Since the serial driver software used in this implementation does not perform concurrent bidirectional transmission, the inter-processor architecture must next be considered. If a connected pair of processors have a master - slave relationship, then only a single software driver is needed to support both input and output to/from the other processor, but if both input and output to/from the other processor can occur concurrently at any time, then a set of drivers and channel link lines will be required. Once the channel number(s) have been selected, the hardware addresses must next be determined.

The next step is to modify the source code of each driver to be used, specifically, the UNITNUMBER, and the channel hardware addresses. If multiple drivers are to be incorporated into the operating system, then each driver must also have its own unique name . Once changes have been made, using the RT-11 operating system and its editor, the drivers must next be re-assembled.

The command to re-assemble a software driver is

```
.R MACRO
* OBJFIL[,LSTFIL]=[OPTIONS,]MACROS,SRCFIL
```

For example,

```
.R MACRO
* DLCH07=ALL1,MACROS,DLCH07
```

The OPTIONS file may set any or all of several assembly-time options to customize the resulting object to a particular hardware configuration. If no OPTIONS file is given, the resulting .OBJ file will run on any PDP/LSI 11 model computer. The option file may include definitions of the following symbols:

EIS	- Causes code to be generated utilizing MUL, ASH, DIV, ASHC, and SOB instructions.
LSI	- Causes code for MTPS, MFPS, and SOB

instructions.

FPI	- Causes code for FADD, FSUB, FMUL, and FDIV instructions. (These are 11/40 type floating point instructions; the 11/45 type instructions are not supported.)
TERAK	- Defines all of the above.

FPI should not be defined unless EIS is defined. For LSI/11s with EISFIS chip, define all three. For LSI/11s without the EISFIS chip, define only LSI. For 11/40s with EIS, define only EIS. For 11/10s, do not define any of the options. For this implementation, ALL1.MAC contains the entries 'EIS=0<CR> LSI=1<CR> FPI=0<CR>'.

Once the new channel link communication software driver(s) have been re-assembled, the next step is to build a new UCSD PASCAL operating system. The following RT-11ish files must be online when the new operating system is to be created (for this particular implementation).

MACROS.MAC	Global definitions and macros for the interpreter sources. These should be assembled in front of all sources, except the boot loaders.
MAINOP.MAC	Interpreter section for most P-machine instructions
PROCOP.MAC	Interpreter section for procedure call operators run-time support subroutines.

IOTRAP.MAC	SYSCOM and trap-vector ASECT and console device driver.
RX.MAC	DEC floppy driver.
LP.MAC	LP11 line printer driver.
SOFTFP.OBJ	Floating point package for systems without FPI instruction set.
DLnn.MAC	DL11 port channel driver(s).
RXBOOT.MAC	Bootloader for RX01 compatible drives.

The remaining files are optional files for other micro-computer architectures.

QX.MAC	Driver for REMEX floppies on TERAK 8510 systems.
RK.MAC	RK05 disk driver.
TK.MAC	8510a screen emulator.
RKBOOT.MAC	Bootloader for RK05 compatible drives.
QXBOOT.MAC	Bootloader for QX type REMEX drives.
HARDFP.MAC	Floating point package for systems with FPI instruction set.
DUMYFP.MAC	A dummy floating point package.

The linking instructions to build a new UCSD PASCAL operating system are as follows.

```
.R LINK
*SYSPDP[,MAP]=IOTRAP,MAINOP,PROCOP,RX,LP/C/B:0
```

*DLCH07,DLCH28,SOFTFP

The first three files must be in the above order. Drivers may be in any order. The floating point information should be last.

Now a new UCSD PASCAL operating system file, called SYSPDP.SAV has been created. The next step is to transfer this file from the RT-11 floppy it was created on, over to a UCSD PASCAL floppy. To accomplish this transfer, a seperate floppy, referred to as the 'transfer floppy', must be used. This floppy should be 'ZEROED' using the PASCAL operating system before the transfer procedure is accomplished. After zeroing the transfer floppy, the following sequence should be accomplished in order.

(1) Under the RT-11 operating system, transfer the newly created SYSPDP.SAV file from the floppy it was created on, over to the transfer floppy.

(2) Request a .DIR/BLOCKS list of the transfer floppy. Record on paper the starting block of where SYSPDP.SAV resides and also the total block size of the file.

(3) Remove the RT-11 system floppy and insert a UCSD PASCAL system floppy. Reboot the system.

(4) Enter the command 'MAKE #5:RT11.TEXT[8]. Since the directory index files for the RT-11 and the UCSD PASCAL operating systems reside in different areas of a floppy, a pascal file must be 'created' over the area where the RT-11 directory resides in order to ensure that the UCSD PASCAL

operating system does not write over it.

(5) Enter the command 'MAKE #5:SYSTEM.PDP-11[nn] where nn equals the recorded size of the SYSPDP.SAV file. This enables the UCSD PASCAL operating system to access the file SYSPDP.SAV created by the RT-11 operating system.

Steps five and six produce a transfer floppy containing dual directories, each having pointers to the same newly created operating system.

(6) Rename the file SYSTEM.PDP11 on the UCSD PASCAL system boot floppy to S.PDP11.

(7) Transfer the file SYSTEM.PDP11 from the transfer floppy over to the system boot floppy.

(8) Reboot the system. The LSI-11 system is now re-configured containing the serial port channel drivers.

Three PASCAL test programs have been included in this appendix. These programs were used to test the new channel drivers.

The first program was used to test the concurrency of two channel drivers performing data transmission to seperate CRTs connected to channels 0 and 2. The successful test of concurrency in data transmission was accomplished at 19.2 Kbaud using UNITWRITE instructions.

The remaining two programs were used to test communication between two LSI-11s connected via a single serial channel link. This test consisted of a message transfer being made back and forth (ping-pong fashion) for 100 iterations. One program first issued a UNITREAD to channel 2 UNITNUMBER 8 while the other program first issued a UNITWRITE to channel 0 UNITNUMBER 7 (link was from channel 2 of machine A to channel 0 of machine B). Each time a message was recieved it was displayed on the machines console. Then a response message was sent out.

```
PROGRAM TEST;   (* TO DRIVE DATA THROUGH 2 CHANNELS *)
TYPE
    DATASTR = STRING[40];
    SENDBUF = PACKED ARRAY [1..1920] OF CHAR;
VAR
    I,J,UNITNBR2,UNITNBR7,UNITNBR8                  : INTEGER;
    MESSAGE1,MESSAGE2,MESSAGE3                      : SENDBUF;
    MSSG01,MSSG02                                   : DATASTR;

BEGIN
    UNITNBR2 := 2;   (* CONSOLE   *)
    UNITNBR7 := 7;   (* CHANNEL 0 *)
    UNITNBR8 := 8;   (* CHANNEL 2 *)

    MSSG01 := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789    ';
    MSSG02 := '0123456789    ABCDEFGHIJKLMNOPQRSTUVWXYZ';

    FOR I := 0 TO 47 DO
       BEGIN
       FOR J := 1 TO 40 DO
          BEGIN
             MESSAGE1 [I * 40 + J] := MSSG01 [J];
             MESSAGE2 [I * 40 + J] := MSSG02 [J];
          END;
       END;

    UNITWRITE (UNITNBR7,MESSAGE1,1920,0,1);

    UNITWRITE (UNITNBR8,MESSAGE2,1920,0,1);

    WRITELN;
    WRITELN ('AS YOU CAN SEE, THIS MESSAGE HITS THE CONSOLE  ');
    WRITELN ('AT THE SAME TIME THE BUFFERED DATA IS SENT TO  ');
    WRITELN ('THE CRTS CONNECTED TO UNITNUMBERS 7(0) AND 8(2)');
    WRITELN;
    WRITELN ('NOW, TYPE IN 20 CHARACTERS (CURSER WILL NOT MOVE)');
    WRITELN;

    UNITREAD  (UNITNBR2,MESSAGE3,20,0,1);
    UNITWAIT  (UNITNBR2);

    WRITELN ('THIS IS THE MESSAGE RECIEVED USING A UNITREAD TO');
    WRITELN ('THE CONSOLE:');
    WRITELN;
    WRITELN (MESSAGE3:20);

    WRITELN;
    WRITELN ('TESTING IS NOW ADJOURNED!..........');
END.
```

```
PROGRAM TESTA;
TYPE
    DATASTR = STRING[40];
    SENDBUF = PACKED ARRAY [1..40] OF CHAR;
VAR
    BUFF001,BUFF002                                     : SENDBUF;
    MSSGAA,MSSGBB                                       : DATASTR;
    RMOT                                                : DATASTR;
    OUTOUT                                              : FILE;
    I,J                                                 : INTEGER;

BEGIN

    MSSGAA := 'THIS DATA IS COMING FROM SYSTEM A A A A ';
    MSSGBB := 'THIS DATA IS COMING FROM SYSTEM B B B B ';
    RMOT   := 'REMOUT:';

    FOR J := 1 TO 40 DO
       BEGIN
          BUFF001 [J] := MSSGAA [J];
          BUFF002 [J] := MSSGBB [J];
       END;

    WRITELN ('START WITH A UNITREAD TO UNITNUMBER 8    ');
    RESET (OUTOUT,RMOT);

    FOR J := 1 TO 100 DO
       BEGIN
          UNITCLEAR (8);
          FOR I := 1 TO 40 DO
             BUFF002[I] := ' ';
          UNITREAD (8,BUFF002,40,0,1);
          WHILE UNITBUSY (8) DO
             BEGIN
                I := I+1;
                I := I-1;
             END;
          WRITELN (J:4,'  ',BUFF002);

          UNITCLEAR (8);
          UNITWRITE (8,BUFF001,40,0,1);
          WHILE UNITBUSY (8) DO
             BEGIN
                I := I+1;
                I := I-1;
             END;
       END;
END.
```

```
PROGRAM TESTB;
TYPE
    DATASTR = STRING[40];
    SENDBUF = PACKED ARRAY [1..40] OF CHAR;
VAR
    BUFF001,BUFF002                                   : SENDBUF;
    MSSGAA,MSSGBB                                     : DATASTR;
    RMOT                                              : DATASTR;
    OUTOUT                                            : FILE;
    I,J                                               : INTEGER;

BEGIN

    MSSGAA := 'THIS DATA IS COMING FROM SYSTEM A A A A ';
    MSSGBB := 'THIS DATA IS COMING FROM SYSTEM B B B B ';
    RMOT   := 'REMOUT:';

    FOR J := 1 TO 40 DO
       BEGIN
          BUFF001 [J] := MSSGAA [J];
          BUFF002 [J] := MSSGBB [J];
       END;

    WRITELN ('START WITH A UNITWRITE TO UNITNUMBER 8    ');
    RESET (OUTOUT,RMOT);

    FOR J := 1 TO 100 DO
       BEGIN
          UNITCLEAR (8);
          UNITWRITE (8,BUFF002,40,0,1);
          WHILE UNITBUSY (8) DO
             BEGIN
                I := I+1;
                I := I-1;
             END;

          UNITCLEAR (8);
          FOR I := 1 TO 40 DO
             BUFF001[I] := ' ';
          UNITREAD (8,BUFF001,40,0,1);
          WHILE UNITBUSY (8) DO
             BEGIN
                I := I+1;
                I := I-1;
             END;
          WRITELN (J:4,' ',BUFF001);
       END;
END.
```

Appendix G

Host Structure Chart Documentation

This appendix consists of two sections documenting the modular design of the host processor's test software. The first section is a list identifying all the modules in the software design. The '*' that appears to the right of the module's title identifys that module as a 'common' module called by more that one calling module. The second section is a set of structure charts showing the interrelated structure of the modules within the subsystem. Note that the filled-in bottom right corner of the module box indicates it is a 'common' module. The module numbers in each section are used as a cross-reference between both documentation sections. The program listings for this subsystem are contained in Volume II of this thesis.

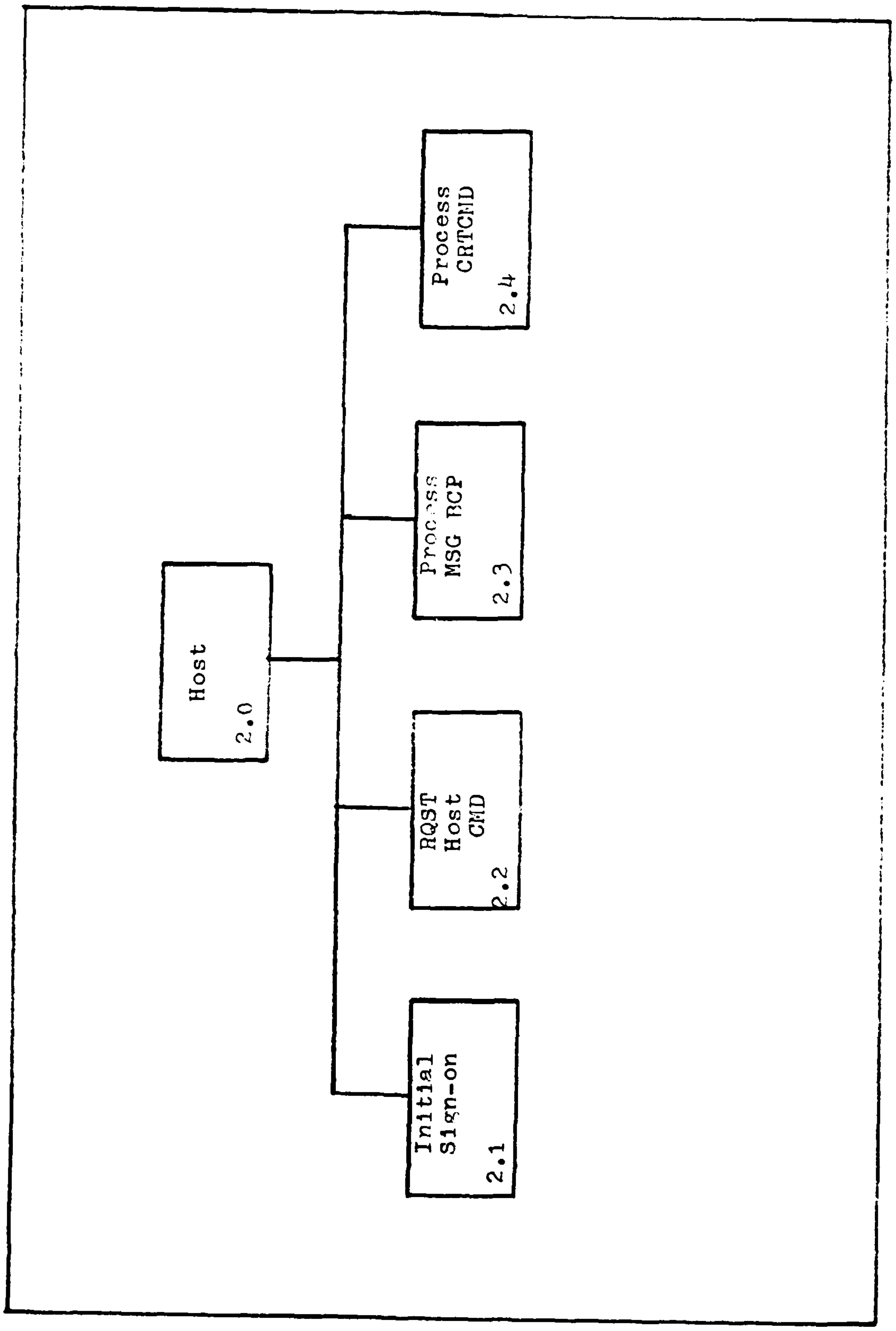
HOST Module List

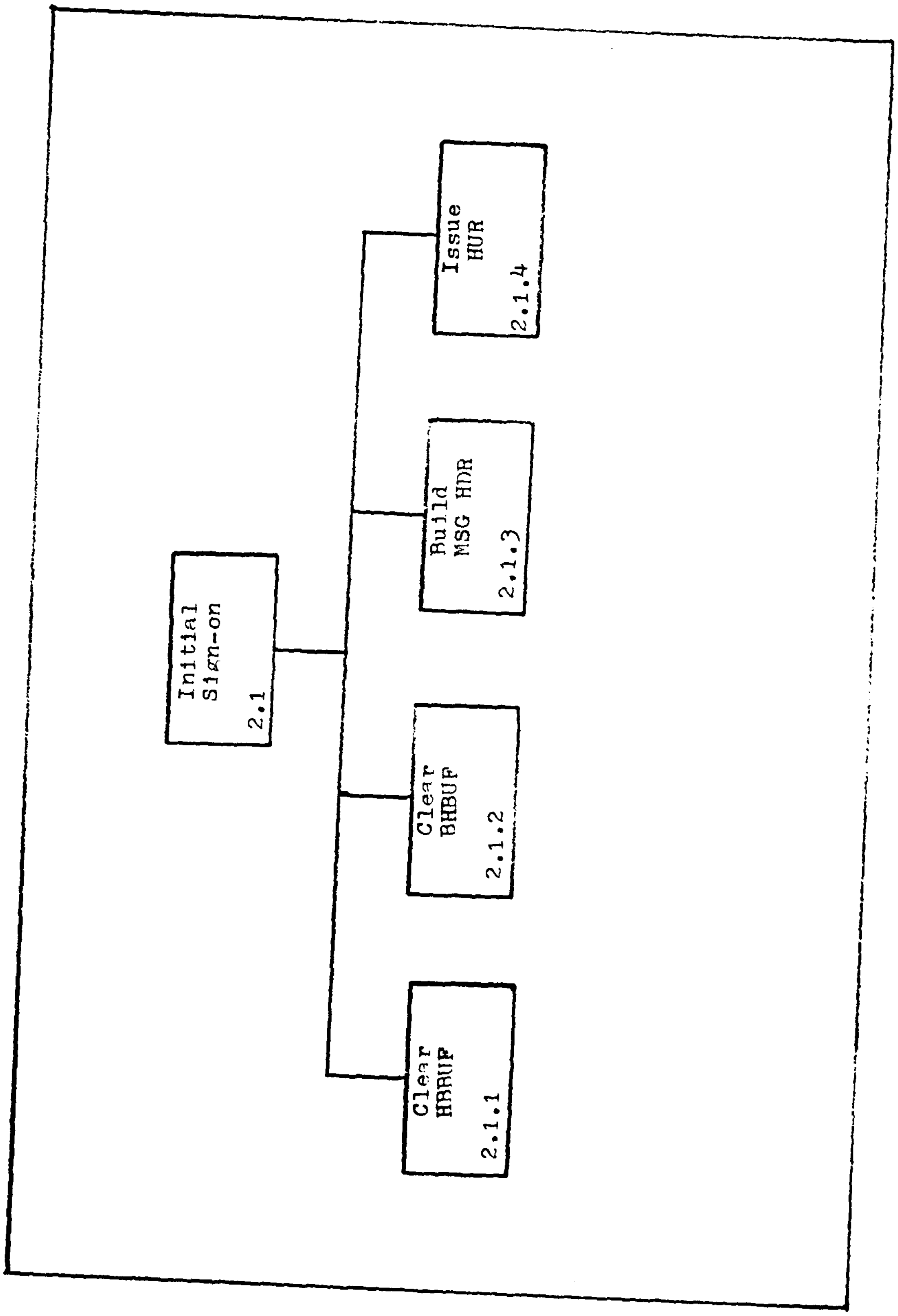
2.0	Host	
2.1	Perform Initial Sign-on to BCP	
2.1.1	Clear the Host-to-BCP Message Buffer	*
2.1.2	Clear the BCP-to-Host Message Buffer	*
2.1.3	Build a Message Header	*
2.1.4	Issue a UNITREAD to the BCP	*
2.2	Request Command from the Host's Console	
2.3	Process the BCP Message Received	

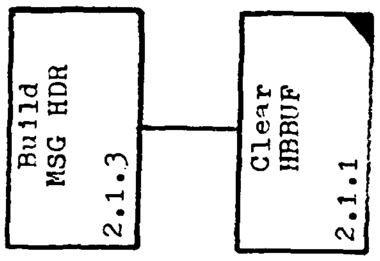
2.4	Process the Host Command Received	
2.4.1	Send Host Request Message to BCP	
2.4.1.1	Increment the Message Number Count	*
2.4.1.2	Issue a UNITWRITE to the BCP	*

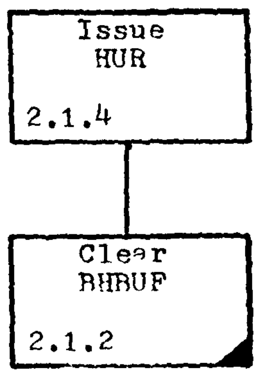
BCP Structure Charts

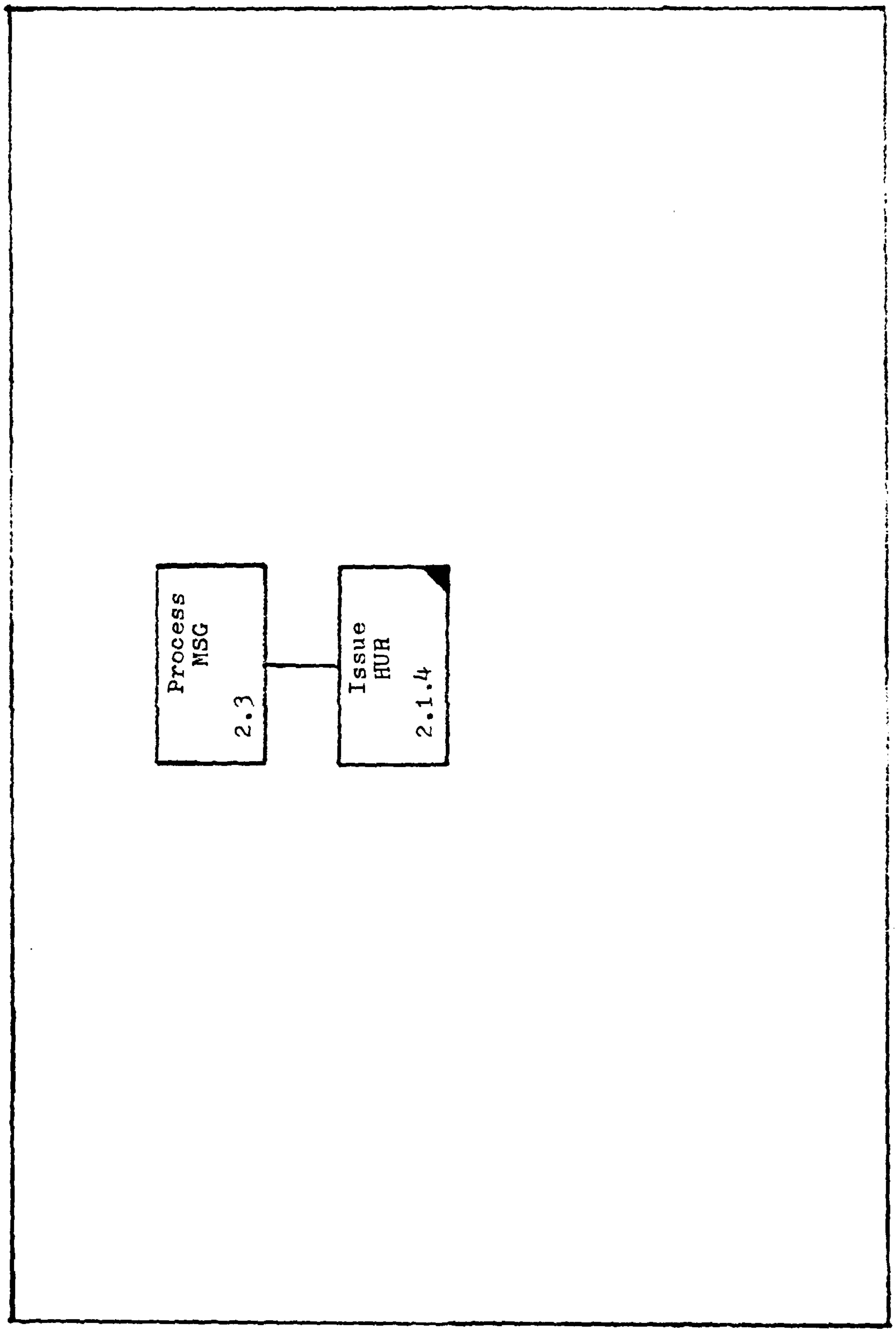
(Charts begin on following page)

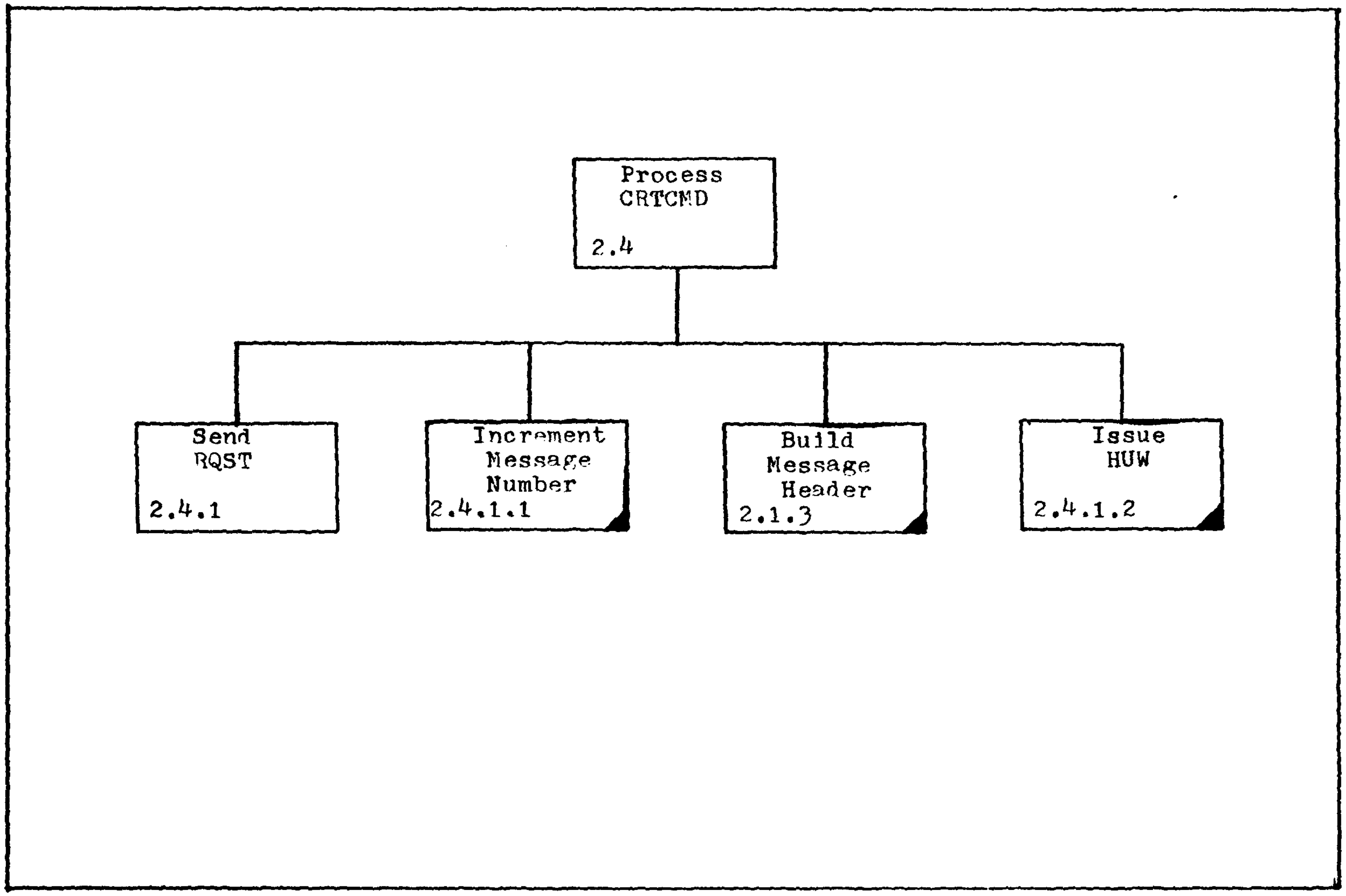

Host
2.0
Initial Sign-on
2.1
RQST Host CMD
2.2
Process MSG RCP
2.3
Process CRTCMD
2.4

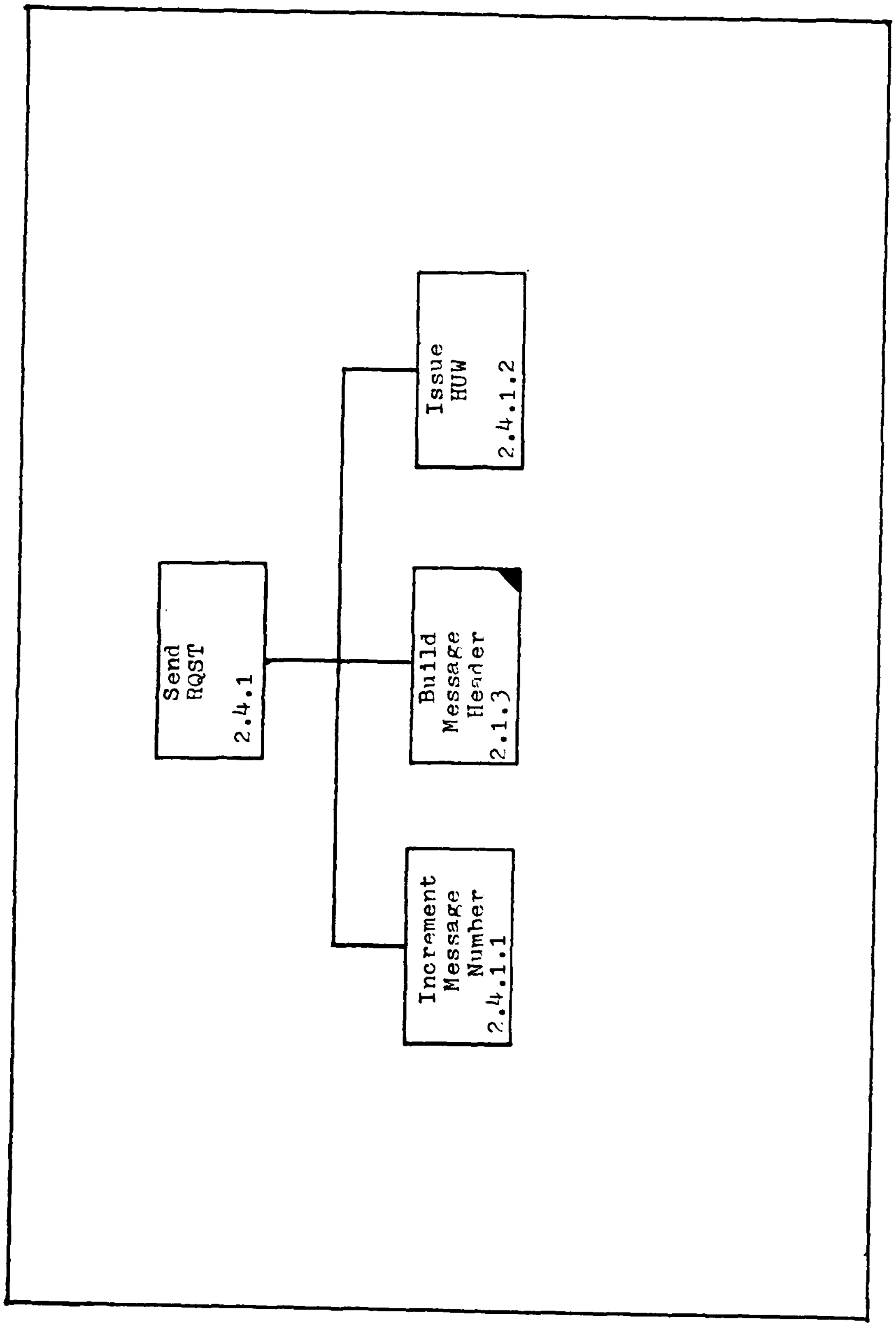

Initial Sign-on
2.1
Clear HBBUF
2.1.1
Clear BHBUF
2.1.2
Build MSG HDR
2.1.3
Issue HUR
2.1.4


Build MSG HDR
2.1.3
Clear HBBUF
2.1.1


Issue HUR
2.1.4
Clear RHBUF
2.1.2


Process MSG
2.3
Issue HUR
2.1.4


Process
CRTCMD
2.4
Send
RQST
2.4.1
Increment
Message
Number
2.4.1.1
Build
Message
Header
2.1.3
Issue
HUW
2.4.1.2


Send RQST 2.4.1
Increment Message Number 2.4.1.1
Build Message Header 2.1.3
Issue HUW 2.4.1.2

Appendix H

QP Structure Chart Documentation

This appendix consists of two sections documenting the modular design of the query processor's software. The first section is a list identifying all the modules in the software design. The '*' that appears to the right of the module's title identifys that module as a 'common' module called by more that one calling module. The second section is a set of structure charts showing the interrelated structure of the modules within the subsystem. Note that the filled-in bottom right corner of the module box indicates it is a 'common' module. The module numbers in each section are used as a cross-reference between both documentation sections.

Of the twelve relational algebra operations that the query processor is designed to process, only the 'select' (3.4.3.1) function module has received further detailed definition; sufficient enough for this stage of the overall development effort.

QP Module List

3.0	QP
3.1	Initialize
3.2	Prepare to Accept a Query Packet Message

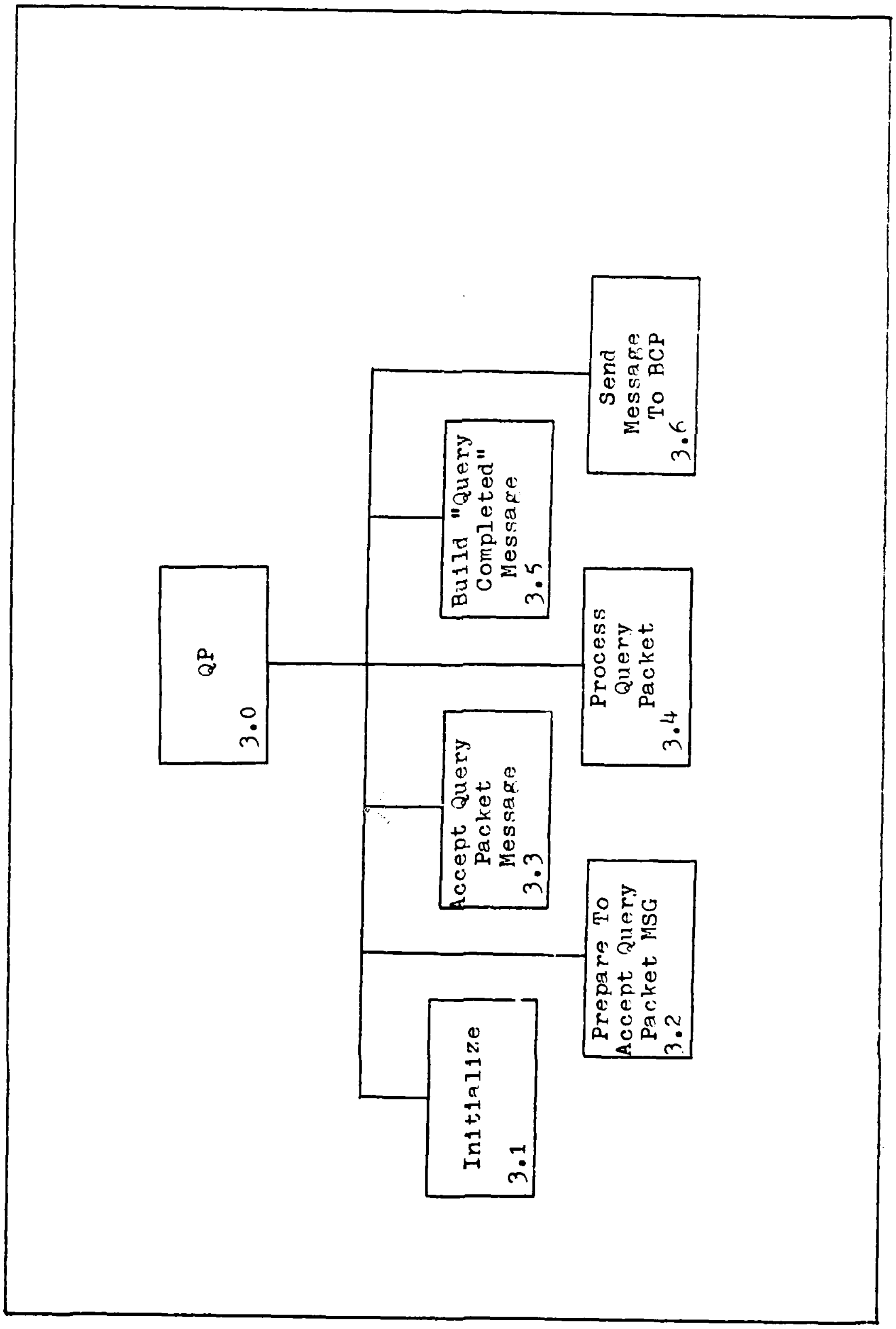
3.3	Accept the Query Packet Message	
3.4	Process the Query Packet Message Request	
3.4.1	Initialize subsystem tables	
3.4.2	Extract Next Query Operation Step	
3.4.3	Process the Query Operation Step	
3.4.3.1	'Select' Operation Processing	
3.4.3.1.1	Decompose the 'Select' Operation	
3.4.3.1.2	Build the 'Get Next Page' Message	
3.4.3.1.3	Send the Message to the BCP	
3.4.3.1.4	Prepare to Accept BCP Message	*
3.4.3.1.5	Read the Response Message	*
3.4.3.1.6	Switch to specified IMM address	*
3.4.3.1.7	Process the Relation Page	
3.4.3.1.7.1	Read a Tuple	*
3.4.3.1.7.2	Perform 'Select' Comparison	
3.4.3.1.7.3	Write Tuple into IMM memory	*
3.4.3.1.7.4	Build an 'Add Page' Message	*
3.4.3.2	'Project' Operation Processing	
3.4.3.3	'Join' Operation processing	
3.4.3.4	'Modify' Operation Processing	
3.4.3.5	'Delete' Operation Processing	
3.4.3.6	'Add' Operation Processing	
3.4.3.7	'Maximum' Operation Processing	
3.4.3.8	'Minimum' Operation Processing	
3.4.3.9	'Count' Operation Processing	
3.4.3.10	'Average' Operation Processing	
3.4.3.11	'Union' Operation Processing	

3.4.3.12 'Destroy' Operation Processing

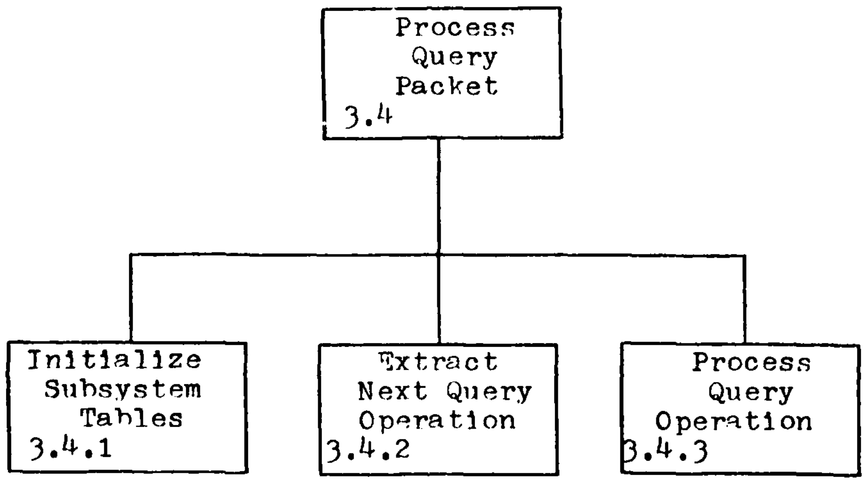
3.5 Build 'Query Processing Completed' Message

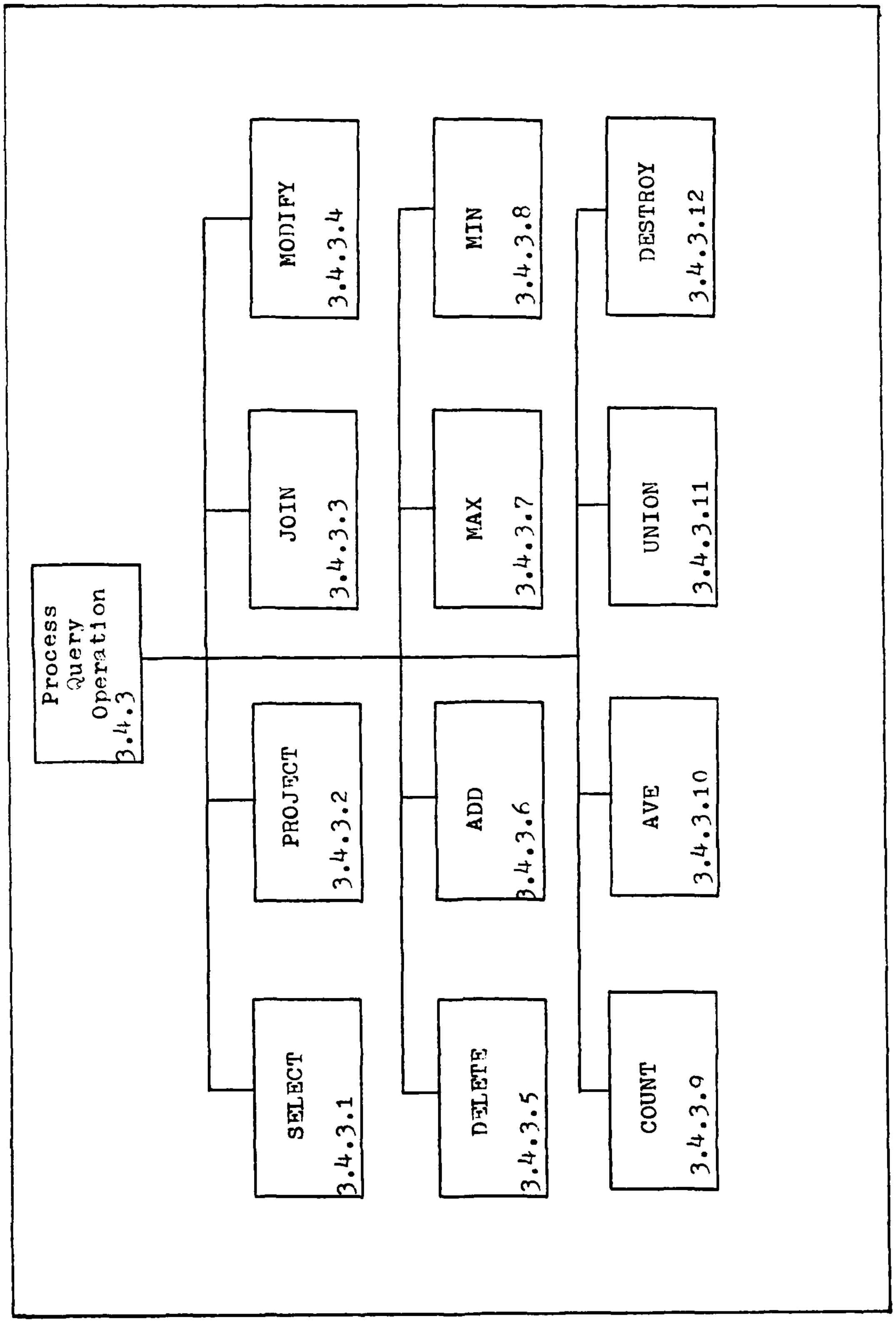
QP Structure Charts

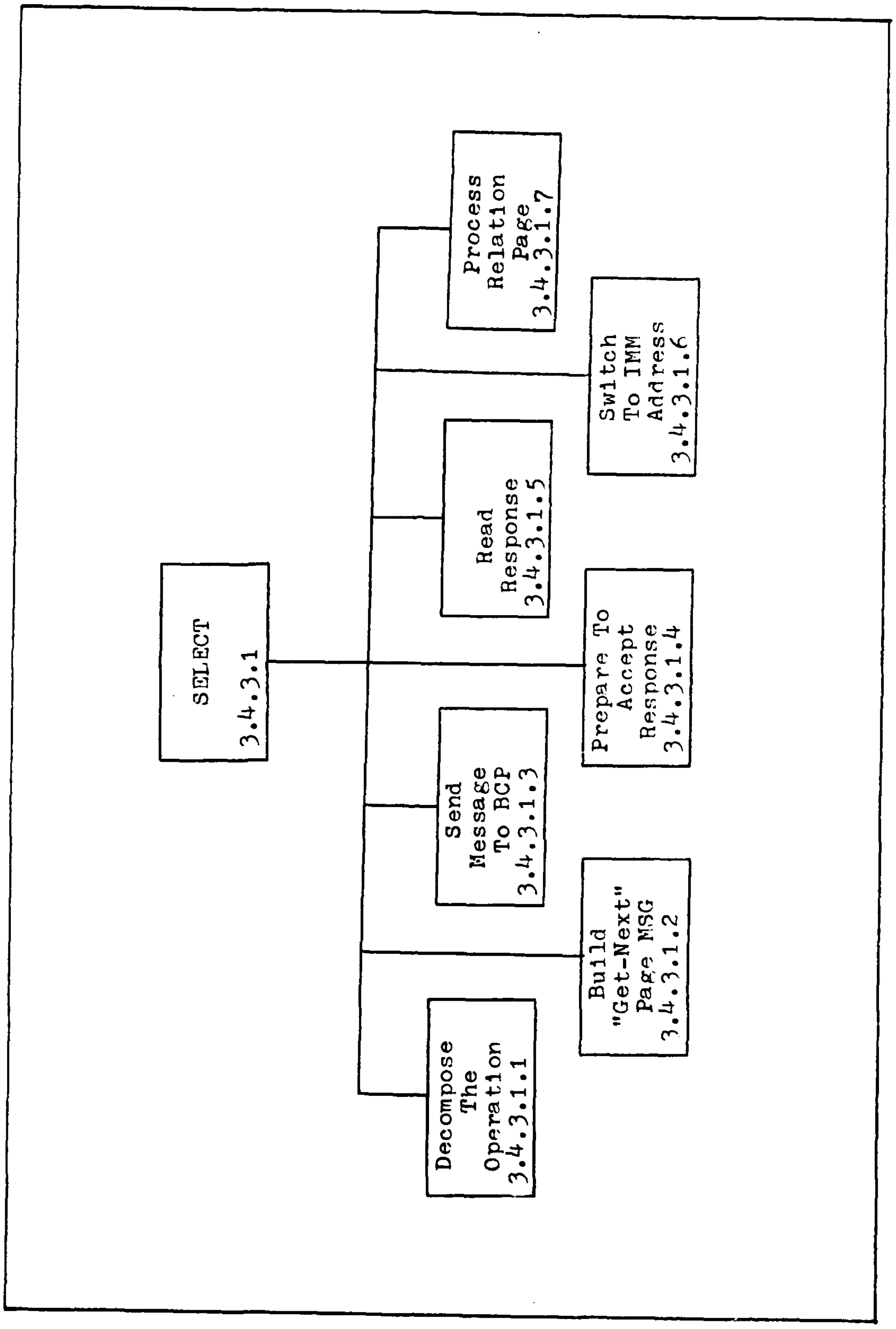
(Charts begin on following page)

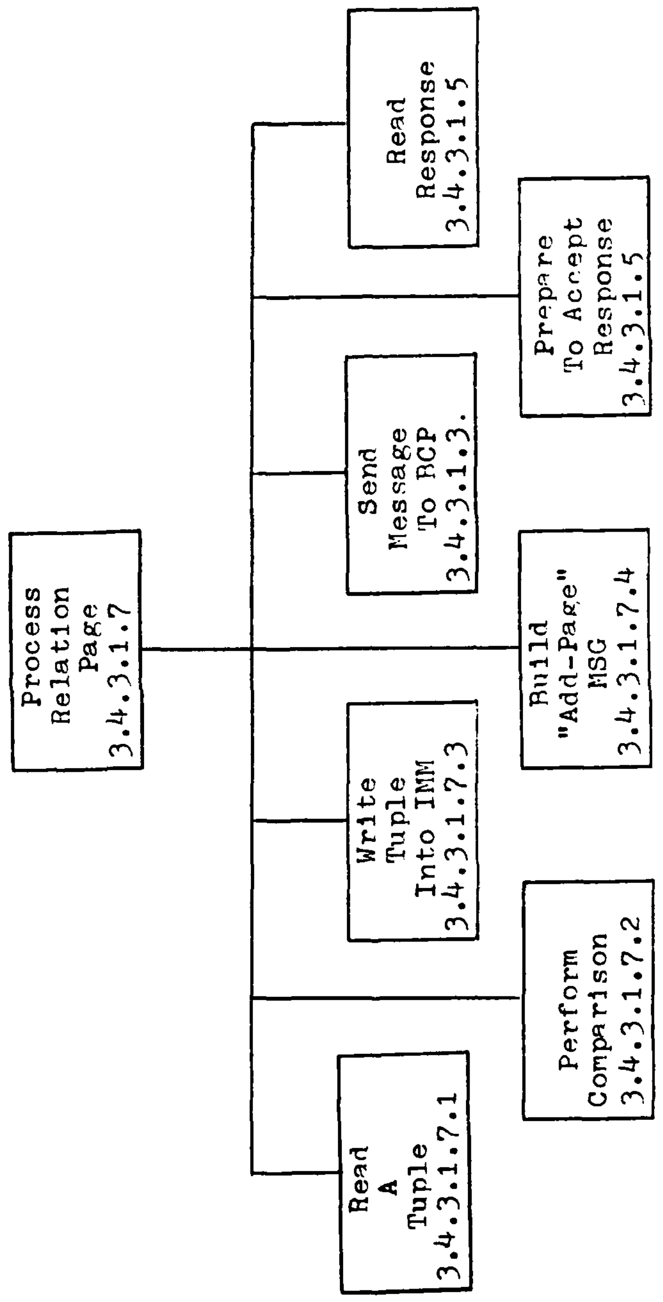


QP 3.0
Initialize 3.1
Prepare To Accept Query Packet MSG 3.2
Accept Query Packet Message 3.3
Process Query Packet 3.4
Build "Query Completed" Message 3.5
Send Message To BCP 3.6


Process Query Packet 3.4
Initialize Subsystem Tables 3.4.1
Extract Next Query Operation 3.4.2
Process Query Operation 3.4.3


Process Query Operation 3.4.3
SELECT 3.4.3.1
PROJECT 3.4.3.2
JOIN 3.4.3.3
MODIFY 3.4.3.4
DELETE 3.4.3.5
ADD 3.4.3.6
MAX 3.4.3.7
MIN 3.4.3.8
COUNT 3.4.3.9
AVE 3.4.3.10
UNION 3.4.3.11
DESTROY 3.4.3.12


SELECT 3.4.3.1
Decompose The Operation 3.4.3.1.1
Build "Get-Next" Page MSG 3.4.3.1.2
Send Message To BCP 3.4.3.1.3
Prepare To Accept Response 3.4.3.1.4
Read Response 3.4.3.1.5
Switch To IMM Address 3.4.3.1.6
Process Relation Page 3.4.3.1.7


Process Relation Page 3.4.3.1.7
Read A Tuple 3.4.3.1.7.1
Perform Comparison 3.4.3.1.7.2
Write Tuple Into IMM 3.4.3.1.7.3
Build "Add-Page" MSG 3.4.3.1.7.4
Send Message To RCP 3.4.3.1.3.
Prepare To Accept Response 3.4.3.1.5
Read Response 3.4.3.1.5

Vita

Captain Robert W. Fonden was born on November 16, 1952, in Houston, Texas. In 1971, he graduated from Marion Senior High School in Marion, Illinois. He attended Moorhead State University, Moorhead, Minnesota from which he received a Bachelor of Arts degree with a major in Computer Science in 1975. He was subsequently commissioned a 2nd Lieutenant in the United States Air Force Reserves. On September 6, 1976 he was called to active duty. Between September 1976 and May 1980, he was assigned to HQ MAC/AD Data Automation at Scott AFB, Illinois, as a computer systems analyst for the Consolidated Aerial Port System (CAPS) development project. He entered the Air Force Institute of Technology in June 1980.

Permanent Address: 195 Richmond Street
St. Paul, MN 56103

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/81D-6	2. GOVT ACCESSION NO. AD-A115 558	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DESIGN AND IMPLEMENTATION OF A BACKEND MULTIPLE PROCESSOR RELATIONAL DATA BASE COMPUTER SYSTEM		5. TYPE OF REPORT & PERIOD COVERED MS Thesis
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert W. Fonden Capt USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE DECEMBER 1981
		13. NUMBER OF PAGES 250
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

15 APR 1982

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

~~FREDRIC C. LYNCH, Maj,~~ USAF
~~Director of Public Affairs~~

[illegible]

Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Relational Data Base Management System
Backend Computer
Data Base Computer
MIMD Computer Architecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A backend multiple-processor relational data base computer system was designed with the goal of implementing a data base management system using state-of-the-art technology. The objective was to overcome the traditional limitations of data base management systems implemented on conventional type computer architectures. Hopefully this would solve the ever-growing problem of information systems becoming obsolete in supporting the growing information needs of the corporate industry.

Toward this goal, investigations were made into studies in the literature involving backend data base computer systems, the relational data model, and data base computers using specialized architectures. The advantages and disadvantages of these three areas were explored and then, after having defined the longterm requirements and goals for the development of such a system, the beneficial characteristics from each of these areas were merged together to produce a system design. Central to this design is the use of a set of processors, managed by a backend controller processor, to take full advantage of three levels of parallelism in processing relational algebra query requests against relations.

Due to the complexity and size of this development effort, a top-down structured detailed design and only a partial implementation of the backend controller processor was achieved in this research effort. A detailed development plan has been defined, consisting of several projected follow-on development efforts, to complete the entire development of this data base computer system.

DATE
ILMED
8